Linux Kernel: Scenarios and Analyses

毛德操　胡希明　著

（上册）

LINUX 内核源代码情景分析

```
 * This should really return information about whether
 * we should do bottom half handling etc. Right now we
 * end up _always_ checking the bottom half, which is a
 * waste of time and is not what some drivers would
 * prefer.
 */
int handle_IRQ_event(unsigned int irq, struct pt_regs * regs,
  struct irqaction * action)
{
  int     status;
  int     cpu = smp_processor_id( );

  irq_enter(cpu, irq);

  status = 1;/* Force the "do bottom halves" bit */

  if(!(action->flags & SA_INTERRUPT))
     __sti( );

  do{
        status |= action->flags;
        action->handler(irq, action->dev_id, regs);
        action = action->next;
  }while (action);
  if(status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
  __cli( );

 irq_exit(cpu, irq);

 return status;
}
```



浙江大學出版社

VISIT…

LANZAROTE Caliente.com

前　　言

本书着重于对 Linux 系统最新版本 (2.4.0) 内核源代码进行情景描述和情景公析。

什么是情影描述？什么是情景分析？

不妨以英语的教学为例。大家都知道，有一种很有效的方法是通过“情景会话”学习英语。例如。去剧院问路要说些什么，去图书馆要说些什么，去餐馆吃饭碰上了熟人又说些什么，等等。每一个这样的“情景”都是一个常见或常用的会话过程。以这样的一些情景为线索，沿着这些线索讲解“这是被动语态“、“那是习惯用法”。就容易引起学习人的兴趣从而印象深刻，并且每学了这样一个情景就能够实际运用。另外，由于来自现实生活的情景在语法、语义等方面都不是单一的，在学习一个情景的时候通常都会涉及该语言种种不同的方面，通过一系列精心安排的情景会话的学习，就能对英语逐步的建立起比较全面的认识。事实上，就英语的学习而言，纯粹的系统化学习方法几乎是不现实的。事实上，很少有人通过读字典来学单词，而都是结合课文来学，每篇课文实际上也是一个情景。当然，系统化的学习还是要的，学习情景对话以后还要再系统的学习语法。但是无可否认的是，从情景对话入手学习英语比从语法入手要有效得多。相信读者会有这方面的体会和经历。

现在来看对 Linux 内核的学习。如果以若干经过精心安排的情景为线索，例如，打开一个文件的全过程，执行一个可执行程序的全过程，从一个进程发送一个报文到另一个进程的过程等等，结合内核源代码逐个加以讲解，并且在讲解过程中有针对性地介绍所涉及的数据结构和算法，读者就能得到对整个内核的生动而深刻的理解。本书的宗旨之一就在于引导读者走过许多这样的“情景”，从而建立起对 Linux 内核的全面的认识。至于情景的安排，仍然按照操作系统的原理分成若干章，例如存储管理、进程管理、文件系统等等。在每一章中，除了必要的叙述以外，都挑选了若干重要的情景，结合源代码逐个加以讲解。

本书所用的源代码，刚开始编写初稿时取自当时最新的 Linux 内核 2.3.38 版，后来历经 2.3.98 和 2.4.0 测试版，最后依据 2.4.0 正式版重新修改定稿。读者可以在相关的网站上自行下载该版内核的全部源代码。可以肯定，当读都看到本书时，甚至本书付印时，最新的版本已不再是 2.4.0 了。但是不管怎么样我们总得要锁定一个版本，这就是 2.4.0。

一般情况下，分析操作系统源代码的专著或教材习惯上都是这样安排的，以主要数据结构的定义为核心，以数据结构之彰的联系为线索，内容则以对文件、模块和函数的功能描述为主，辅以若干函数中的代码片断作为实例，以达到介绍、分析各种特定机制的目的。这种思路和安排基本上类似于先讲语法规则后举一些例句的外语教学方法，它比较适合于只要求对内核和它的原理有粗略了解的读者，但对需要深入理解内核或实际从事这方面工作的读者就未必合适。其实，这种安排对于初学都也未必是最好的。不错，要理解一个操作系统的内在机制及其实现机理，当然需要了解主要数据结构的组成，了解数据结构之间的联系，了解整个内核代码的模块划分、文件划分和功能分解，了解主要函数对有关数据结构操作的大致逻辑流程。问题在于，怎样才能使读者和学生达到这些要求。根据我们多年来的切身体会，我们决定从具体、鲜活的源代码入手作情景分析，在分析过程中逐步引入相关的数据结构和互相间的联系，介绍具体函数的逻辑流程及其物理背景乃至代码作者的某些高超技巧，让读者和作者一起完成必要的抽象过程，通过读者的思索，最后达到深入而全面的理解。

对于从事系统设计或实现的读者，源代码的阅读和理解是一项重要的基本功。写小说的人大多的读了许多名著和文学评论以后，而不是读了“小说概论”以后才学到写作技巧，进而写出受读者喜爱的作品。写程序的人又何尝不是如此。本书的目的之一就是为读者提供一些类似于文学评论的材料。

别一方面，源代码的阅读和理解也是必要的。在某种意义上，源代码本身既是最准确的说明书也是最权威的教科书，因为由它所构成的系统切切实实在运行。我们自己就有过这样的经历：学了一些原理和抽象的流程就自以为懂了，可是拿源代码一看却怎么也对不上号。于是下决心钻进去，花了九牛二虎之力才搞懂。Linux 内核源代码还为计算机行业的工作人员树立了一个参照物。我们在工作中常常看到，人们(包括我们自己)在碰到问题时往往会先想一想：这在 Linux(以前是 Unix)里面是怎样实现的？或者在 Linux 环境中能否实现？再查看一下有关的源代码，便有了主张。有时甚至就在源代码中找几个文件加以裁剪、修改，问题很快就解决了（但须遵守 GPL 中的有关规定）。诚然，Linux 内核源代码的阅读和理解是个艰苦的过程，最好能有些指导，有些帮助，而这正是我们写作本书的目的。

希望读者在每读完一章后能做两个小结。一个是关于数据结构组成和数据结构之间联系的小结，另一个是关于执行过程以及函数调用关系的小结。读者为了完成这两个小结，可能不得不回过头去再读一遍甚至几遍前面的内容。从内容的选定和编排的角度来说，最理想的当然是严格遵循“先说明后引用”的原则，像平面几何那样建立起一个演绎体系。可是，对于一个实际的系统，特别是对于它的源代码，这种完全线性的叙述和认识过程是不现实的。实际的认识过程是螺旋式的，这也决定了常常需要反复读几遍才能理解。所以，对于一个操作系统的源代码，读到后面再返回前面，再读到后面又返回前面，这几乎是必然的过程。真有决心深入了解 Linux 内核的读者应该有这个思想准备。我们相信，读者在读完全书以后，如果闭目细想，一定会有一种在一个新到的城市中由向导陪同走街串巷，到过了大量的重要景点，最后到了某个高楼之顶的旋转餐厅鸟瞰整个城市是时常会有的那种心情。

由于篇幅的原因，全书分上下两册。上册包括预备知识、存储管理、中断和系统调用、进程和进程调度、文件系统以及传统的 Unix 进程间通讯，共六章。下册则分基于 socket 的进程间通讯、设备驱动、多处理器 SMP 系统结构以及系统引导和初始代，共四章。上下两册不可分割，是一个有机的整体。

本书的题材闷决定了读者主要是中、高级的计算机专业人员，以及大学有关专业的高年级学生和研究生。但是，我们在写作中也尽量照顾到了非计算机专业的学生和初学者（因此程序高的读者有时候也许会觉得书中有些讲解的过于啰嗦）。一般而言，读者只要有一些操作系统和计算机系统结构方面的基础知识，并粗通 C 语言，就可以阅读本书。

就像软件免不了有错一样，对软件的理解和诠释也一定会有错误，我们能做的只是尽量减少错误。我们可以负责地说，本书付印前在文字中已经没有我们知道而没有改正的错误，更没有故意误导读者的内容。但是，我们深知错误一定是有的，我们欢迎讨论，欢迎批评。

20 年前，本书的两位作者从不同单位调入浙江大学计算机系，共事期间曾共同承担过若干计算机应用项目的开发、研究。后来第一作者去了美国，目前在美定居，继续从事计算机专业的工作；第二作者已从学校退休，目前受聘在杭州恒生电子股份有限公司任职。出于难忘的友情和其他一些难以割舍的情结（包括对 Unix 和 Linux 的共同爱好），两年前通过越洋电话商定要合作写几本有关 Linux 的书，此刻在读者手上的就是其中的第一本，其余的就要看条件是否允许了。

从成书到出版，曾得到了陈大中、曾抗生、金通洸、俞瑞钊几位教授和其他许多国内朋友的鼓励和支持，作者感谢他们。特别要提到的是，恩师何志钧教授在过去和现在都给了作者许多的关心和帮助，令作者终生难忘，本书的出版在某种意义上也是对恩师的一次汇报，同时表示由衷的感谢。

作者还要感谢恒生电子股份有限公司黄大成总裁、彭政纲董事长以及其他领导人对作者，特别

是第二作者开展 Linux 研究的支持。他们为本书第二作者提供了很好的工作条件，使其在工作之余继 Unix 之后还能再从事 Linux 技术的研究。作为全国著名软件企业的决策者，他们对软件核心技术的敏锐眼光以及采用最新技术为我国金融证卷行业开发全新的大型应用软件的战略决策，令人软佩。感谢他们，祝愿他们取得更大的成功。

最后，还要感谢谢敏、王红女、章西、李清瑜几位小姐，她们在承担公司繁重工作的同时利用业余时间为本书文稿的录入和整理付出了大量的辛勤劳动。

本书的出版，像任何其他技术专著一样，除了错误之外总还会有许多不尽人意的地方，欢迎国内外的专家和本书读者给我们指出，以便改进。

毛德操、胡希明

2001 年 5 月 1 日

目　　录

第1章　预备知识……………………………………………………………………1
1.1　Linux 内核简介……………………………………………………………..1
1.2　Intel X86 CPU 系列的寻址方式………………………………………...…...6
1.3　i386 的页式内存管理机制…………………………………………………..12
1.4　Linux 内核源代码中的 C 语言代码………………………………………...15
1.5　Linux 内核源代码中的汇编语言代码………………………………………21

第2章　存储管理…………………………………………………………………..29
2.1　Linux 内存管理的基本框架……………………………………………….....29
2.2　地址映射的全过程……………………………………………………...….33
2.3　几个重要的数据结构和函数…………………………………………...….40
2.4　越界访问……………………………………………………………………...53
2.5　用户堆栈的扩展……………………………………………………………...57
2.6　物理页面的使用和周转……………………………………………...….66
2.7　物理页面的分配………………………………………………………………...77
2.8　页面的定期换出……………………………………………………………...90
2.9　页面的换入…………………………………………………………………….125
2.10　内核缓冲区的管理………………………………………………………….131
2.11　外部设备存储空间的地址映射…………………………………………….154
2.12　系统调用 brk()……………………………………………………………….160
2.13　系统调用 mmap()……………………………………………………………...178

第3章　中断、异常和系统调用…………………………………………………191
3.1　X86 CPU 对中断的硬件支持…………………………………………….....192
3.2　中断向量表 IDT 的初始化………………………………………………......196
3.3　中断请求队列的初始化…………………………………………………….203
3.4　中断的响应和服务………………………………………………………….209
3.5　软中断与 Bottom Half……………………………………………………….221
3.6　页面异常的进入和返回……………………………………………………..233
3.7　时钟中断……………………………………………………………………..236
3.8　系统调用……………………………………………………………………..244
3.9　系统调用号与跳转表………………………………………………………..256

第 4 章 进程与进程调度……………………………………………………………263
4.1 进程四要素……....………………………………………………………….263
4.2 进程三部曲：创建、执行和消……………………………………………….267
4.3 系统调用 fork()、vfork()与 clone()………………………………………...280
4.4 系统调用 execve()…………………………………………………………...305
4.5 系统调用 exit()与 wait4()…………………………………………………...337
4.6 进程的调度与切换………………………………………………………….356
4.7 强制性调度………………………………………………………………….377
4.8 系统调用 nanosleep()和 pause()…………………………………………….384
4.9 内核中的互斥操作………………………………………………………….398

第 5 章 文件系统……………………………………………………………………415
5.1 概述………………………………………………………………………….415
5.2 从路径名到目标节点……………………………………………………….431
5.3 访问权限与文件安全性…………………………………………………….469
5.4 文件系统的安装的拆卸…………………………………………………….491
5.5 文件的打开与关闭………………………………………………………….540
5.6 文件的写与读……………………………………………………………….579
5.7 其他文件操作……………………………………………………………….640
5.8 特殊文件系统 /proc………………………………………………………...653

第 6 章 传统的 Unix 进程间通信……………………………………………………..687
6.1 概述………………………………………………………………………….687
6.2 管道和系统调用 pipe()……………………………………………………..689
6.3 命名管道…………………………………………………………………….709
6.4 信号………………………………………………………………………….716
6.5 系统调用 ptrace()和进程跟踪……………………………………………….756
6.6 报文传递…………………………………………………………………….773
6.7 共享内存…………………………………………………………………....802
6.8 信号量……………………………………………………………………….829

第 7 章 基于 socket 的进程间通信……………………………………………………843
7.1 系统调用 socket()……………………………………………………………...843
7.2 函数 sys_socket()——创建插口…………………………………………………852
7.3 函数 sys_bind()——指定插口地址……………………………………………...867
7.4 函数 sys_listen()——设定 server 插口…………………………………………..877
7.5 函数 sys_accept()——接受连接请求……………………………………………879
7.6 函数 sys_connect()——请求连接………………………………………………..890
7.7 报文的接收与发送…………………………………………………………………907
7.8 插口的关闭…………………………………………………………………………943
7.9 其它…………………………………………………………………………………958

第 8 章 设备驱动…………………………………………………………………………….960
8.1 概述…………………………………………………………………………………960
8.2 系统调用 mknod()………………………………………………………………...968
8.3 可安装模块………………………………………………………………………..975
8.4 PCI 总线……………………………………………………………………………1018
8.5 块设备的驱动……………………………………………………………………1086
8.6 字符设备驱动概述………………………………………………………………1163
8.7 终端设备与汉字信息处理………………………………………………………1166
8.8 控制台的驱动……………………………………………………………………1198
8.9 通用串行外部总线 USB………………………………………………………...1249
8.10 系统调用 select()以及异步输入/输出…………………………………………..1400
8.11 设备文件系统 devfs……………………………………………………………..1419

第 9 章 多处理器 SMP 系统结构…………………………………………………….1437
9.1 概述……………………………………………………………………………1437
9.2 SMP 结构中的互斥问题……………………………………………………...1441
9.3 高速缓存与内存的一致性……………………………………………………1445
9.4 SMP 结构中的中断机制……………………………………………………...1453
9.5 SMP 结构中的进程调度……………………………………………………...1466
9.6 SMP 系统的引导……………………………………………………………...1472

第 10 章 系统引导和初始化……………………………………………………………......1492
10.1 系统引导过程概述………………………………………………………………..1492
10.2 系统初始代（第一阶段）…………………………………………………………..1496
10.3 系统初始化（第二阶段）…………………………………………………………..1512
10.4 系统初始化（第三阶段）…………………………………………………………..1548
10.5 系统的关闭和重引导……………………………………………………………..1566

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第1章 预备知识

1.1 Linux 内核简介

在计算机技术的发展史上，Unix 操作系统的出现是一个重要的里程碑。早期的 Unix 曾免费供美国及一些西方国家的大学和科研机构使用，并且提供源代码，这一方面为高校和科研机构普及使用计算机提供了条件；另一方面，也是更重要的，为计算机软件的核心技术“操作系统”的教学和实验提供了条件。特别是 Unix 内核第 6 版的源代码，在相当长的一段时期内是大学计算机系高年级学生和研究生使用的教材，甚至可以说，美国当时整整一代的计算机专业人员都是读着 Unix 的源代码成长的。反过来，这也促进了 Unix 的普及和发展，并且在当时形成了一个 Unix 产业。事实上，回顾硅谷的形成和发展，也可以看到 Unix 起着重要的作用。Unix 两大主流之一的 BSD 就是在加州大学伯克利分校开发的。后来，Unix 成了商品，其源代码也受到了版权的保护，再说也日益复杂和庞大了，而第 6 版则又慢慢显得陈旧了，便逐渐不再用 Unix 内核的源代码作为教材了（但是直到现在还有在用的）。

在这种情况下，出于教学的需要，荷兰的著名教授 Andrew S. Tanenbaum 编写了一个小型的“类 Unix”操作系统 Minix，在 PC 机上运行，其源代码在 20 世纪 80 年代后期和 90 年代前期曾被广泛采用。但是，Minix 虽说是“类 Unix”，其实离 Unix 相当远。首先，Minix 是个所谓“微内核”，与 Unix 内核属于不同的设计，功能上更是不可同日而语。再说 Unix 也不仅仅是内核，还包括了其“外壳 ”Shell 和许多工具性的“实用程序”。如果内核提供的支持不完整，就不能与这些成分结合起来形成 Unix 环境。这样，Minix 虽然不失为一个不错的教学工具，却缺乏实用价值。看到 Minix 的这个缺点，当时的一个芬兰学生 Linus Torvalds 就萌生了一个念头，即组织一些人，以 Minix 为起点，基本上按照 Unix 的设计，并且博采各种版本之长，在 PC 机上实现，开发出一个真正可以实用的 Unix 内核。这样，公众就既有免费的（现代）Unix 系统，又有系统的源代码，且不存在版权问题。可是，Tanenbaum 教授的目光却完全盯在教学上，因此并不认为这是一个好主意，没有采纳这个建议。

毕竟是“初生牛犊不怕虎”，加上自身的天赋和勤奋，还有公益心，Linus Torvalds 就自己动手干了起来。由于所实现的基本上是 Unix，Linus Torvalds 就把它称为 Linux。那时候互联网虽然还不像现在这么普及，但是在大学和公司中已经用得很多了。Linus Torvalds 在基本完成了 Linux 内核的第一个版本以后就把它放在了互联网上，一来是把自己写的代码公诸于众，二来是邀请有兴趣的人也来参与。他的这种做法很快便引起了热烈的反应，并且与美国“自由软件基金会 ” FSF 的主张正好不谋而合。当时 FSF 已经有计划要开发一个类 Unix（但又不是 Unix，所以称为 GNU，这是“ Gnu is Not Unix”的缩写）的操作系统和应用环境，而 Linux 的出现正是适得其时，适得其所。于是，由 Linus Torvalds 主持的 Linux 内核的开发、改进与维护，就成了 FSF 的主要项目之一。同时，FSF 的其它项目，如 GNU 的 C 编译 gcc、程序调试工具 gdb，还有各种 Shell 和实用程序，乃至 Web 服务器 Apache、浏览器 Mozilla（实际上就是 Netscape）等等，则正好与之配套成龙。人们普遍认为自由软件的开发是软件领域中的一个奇迹。这么多志愿者参与，只是通过互联网维持松散的组织，居然能有条不紊地互相配合，开发

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

出高质量的而且又是难度较大的系统软件，实在令人赞叹。

那么，Linux 与它的前身 Minix 的区别何在呢？简单地说，Minix 是个“微内核”，而 Linux 是个“宏内核”；Minix 是个类 Unix 的教学用模型，而 Linux 基本上就是 Unix，而且是 Unix 的延续和发展，甚至是各种 Unix 版本与变种的集大成者。

大家知道，传统意义下的操作系统，其内核应具备多个方面的功能或成分，既包含用于管理属于应用层的“进程”的成分，如进程管理，也包含为这些进程提供各种服务的成分，如进程间通信、设备驱动和文件系统等等。内核中提供各种服务的成分与使用这些服务的进程之间实际上就形成一种典型的“Client/Server”的关系。其实，这些服务提供者并不一定非得都留在内核中不可，他们本身也可以被设计并实现某些“服务进程”，其中必须要留在内核中的成分其实只有进程间通信。如果把这些服务提供者从内核转移到进程的层次上，那么内核本身的结构就可以大大减小和简化。而各个服务进程，既然已从内核中游离出来，便可以单独地设计，实现以及调试，更重要的是可以按实际的需要来配置和启动。基于这样的想法，各种“微内核”（Micro-Kernel）便应运而生。特别是对于一些专用的系统，主要是实时系统和“嵌入式”系统（Embedded System），微内核的思想就很有吸引力。究其原因，主要是因为通常这些系统都不带磁盘，整个系统都必须放在 EPROM 中，常常受到存储空间的限制，而所需要的服务又比较单一和简单。所以，几乎所有的嵌入式系统和实时系统都采用微内核，如 PSOS，VxWorks 等。当然，微内核也有缺点，将这些服务的提供都放在进程层次上，再通过进程间通信（通常是报文传递）提供服务，势必增加系统的运行开销，降低了效率。

与微内核相对应，传统的内核结构就称为“宏内核”（Macro-Kernel），或称为“一体化内核”（Monolithic Kernel）。通用式的系统由于所需的服务面广而量大，一体化内核就更为合适，作为一种通用式系统，Linux 采用一体化内核是很自然的事。

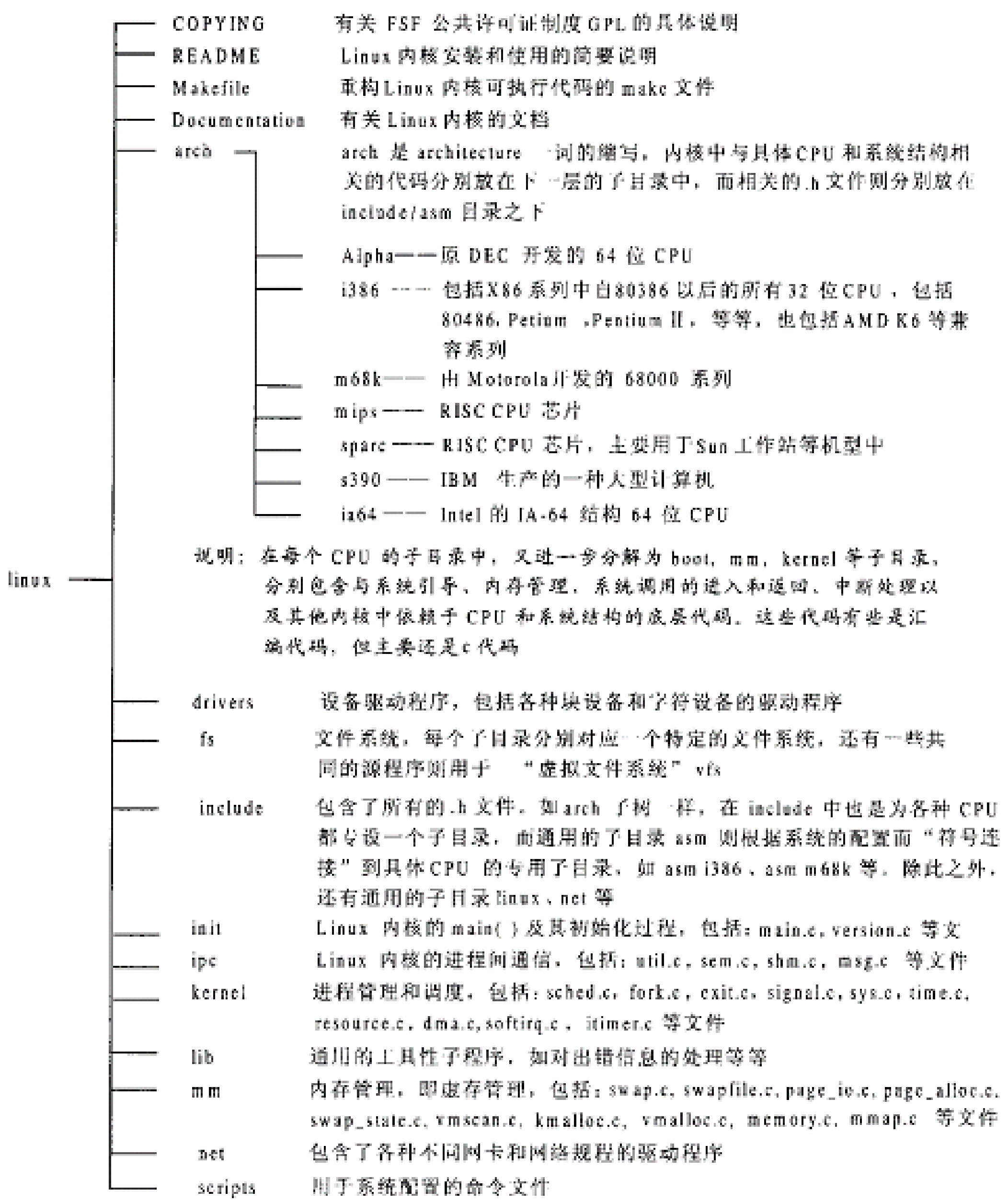
传统的 Unix 内核是“全封闭”的。如果要往内核中加一个设备（增加一种服务），早期一般的做法是编写这个设备的驱动程序，并变动内核源程序中的某些数据结构（设备表），再重新编译整个内核，并重新引导整个系统。这样做当然也有好处，如系统的安全性更能得到保证，但其缺点也是很明显的，那就是太僵化了。在这样的情况下，当某一个公司开发出一种新的外部设备时（比方说，一台彩色扫描仪），它就不可能随同这新的设备提供一片软盘或光盘给用户，使得用户只要运行一下“setup”就可以把这设备安装上了（像对 DOS/Windows 那样），有能力修改 Linux 内核的设备表，并重新编译内核的用户毕竟不多。

在 Linux 里，这个问题就解决得比较好。Linux 既允许把设备驱动程序在编译时静态地连接在内核中，一如传统的驱动程序那样；也允许动态地在运行时安装，称为“模块”；还允许在运行状态下当需要用到某一模块时由系统自动安装。这样的模块仍然在内核中运行，而不是像在微内核中那样作为单独的进程运行，所以其运行效率还是得到保证。模块，也就是动态安装的设备驱动程序的实现（详见设备驱动程序一章），是很大的改进。它使 Linux 设备驱动程序的设计、实现、调试以及发布都大大地简化，甚至可以说是发生了根本性地变化。

Linux 最初是在 Intel 80386“平台”上实现的，但是已经被移植到各种主要的 CPU 系列上，包括 Alpha、M68K、MIPS、SPARC、Power PC 等等（Pentium、Pentium II 等等均属于 i386 系列）。可以说 Linux 内核是现今覆盖面最广的一体化内核。同时，在同一个系列的 CPU 上，Linux 内核还支持不同的系统结构，它既支持常规的单 CPU 结构，也支持多 CPU 结构。不过，本书将专注于 i386 CPU，并且以单 CPU 结构为主，但是最后有一章专门讨论多 CPU 结构。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在安装好的 Linux 系统中，内核的源代码位于/usr/src/linux。如果是从 GNU 网站下载的 Linux 内核的 tar 文件，则展开以后在一个叫 linux 的子目录中。以后本书中谈到源文件的路径时，就总是从 linux 这个节点开始。Linux 源代码的组成，大体如下所示。



值得一提的是，Linux 的源代码看似庞大，其实对于每一个具体的内核而言并不是所有的.c 和.h 文件都会用到，而是在编译（包括连接）时根据系统的配置有选择地使用。例如，虽然源代码中包含了用来支持各种不同 CPU 的代码，但编译以后每一个具体的内核都只是针对一种特定 CPU 的。再如，在 net 子目录下包含了各种网卡的驱动程序，但实际上通常只会用到一种网卡，而且各种网卡的驱动程

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

序实际上大同小异。

在结束本节之前，还要介绍一下有关 Linux 内核版本的一些规定。

通常，在说到 Linux 时，是指它的内核加上运行在内核之上的各种管理程序和应用程序。严格地说，内核只是操作系统的一部分，即其核心部分。但是，人们往往把 Linux 的内核就称为 Linux，所以在讲到 Linux 时有时候是指整个操作系统，有时候则是指其内核，要根据上下文加以区分。在本书中，如无特别说明，则 Linux 通常是指其内核。

Linux 内核的版本在发行上有自己的规则，可以从其版本号加以识别。版本号的格式为“ x.yy.zz”。其中 x 介于 0 到 9 之间，而 yy、zz 则介于 0 到 99 之间。通常数字愈高便说明版本愈新。一些版本号后面有时会见到 pNN 的字样，NN 是介于 0 到 20 之间的数字。它代表对某一版的内核“打补丁”或修订的次数。如 0.99p15，代表这是对版本 0.99 的内核的第 15 次修订。

由于 Linux 源代码的开放性，公众随时都可以从网上下载最新的版本，包括还在开发中、尚未稳定、因而还不能发行的版本，因此，需要有一套编号的方案，使用户看到一个具体的版本号就可以知道是属于“发行版”还是“开发版”，所以 Linux 内核的版本编号是有规则的。在版本号 x.yy.zz 中，x 的不同号码标志着内核在设计上或实现上的重大改变，yy 一方面表示版本的变迁，一方面标志着版本的种类，即“发行版”或“开发版”。如果 yy 为偶数便表示是一个相对稳定、已经发行的版本；若为奇数则表示还在开发中，目前还不太稳定、或者在运行中可能出现比较大的问题的版本。开发中的版本一旦通过测试以及试运行，证明已经稳定下来，就可能会发布一个 yy 的值为偶数的发行版。之后，开发者们又将创建下一个新的开发版本。但是有时候也会在历经了几个开发版以后才发布一个发行版。至于 zz，则代表着在内核增加的内容不是很多、改动不是很大时的变迁，只能算是同一个版本。例如，版本由 2.0.34 升级到 2.0.35 只意味着版本 2.0.34 中的一些小缺陷被修复，或者代码有了一些小的改变。“发行版”和“开发版”的 zz 是独立编号的，因此并没有固定的对应关系。例如，当开发版 2.3 的版本号达到 2.3.99 时，相应的发行版还只是 2.2.18。

Linux 内核的 0.0.2 版在 1991 年首次公开发行，2.2 版在 1999 年 1 月发行。Linux 内核的改进是相当频繁的，几乎每个月都在变。本书最初采用的是 2.3.28 版，最后成书付印时则以正式发行的 2.4.0 版为依据。

Linux 的内核基本上只有一种来源，那就是由 Linus 主持开发和维护的内核版本。但是有很多公司在发行 Linux 操作系统不同的发行版（distribution），如 Red Hat、Caldera 等等。虽然不同的发行版本中所采用的内核在版本上有所不同，但其来源基本一致。各发行版的不同之处一般表现在安装程序、安装界面、软件包的多少、软件包的安装和管理方式等方面，在特殊情况下也有对内核代码稍作修改的（如汉化）。不同的发行版由不同的发行商提供服务。不同的发行商对自己所发行版本的定位也有不同，各厂商所能提供的售后服务、技术支持也各不相同。由此可见，原则上全世界只有一个 Linux，所谓“某某 Linux”只是它的一种发行版本或修订版本。另外，不要把 Linux 内核的版本与发行商自己的版本（如“ Red Hat 6.0”）混淆，例如，Caldera 2.2 版的内核是 2.2.5 版。

对于大多数用户，由发行商提供的这些发行版起着十分重要的作用。让用户自行配置和生成整个系统是相当困难的，因为那样用户不但要自己下载内核源程序，自己编译安装，还要从不同的 FTP 站点下载各种自由软件添加到自己的系统中，还要为系统加入各种有用的工具，等等。而所有这些工作都是很费时费力的事情。Linux 的发行厂商正是看到了这一点，替用户做了这些工作，在内核之上集成了大量的应用软件。并且，为了安装软件，发行厂商同时还提供了用于软件安装的工具性软件，以利

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

于用户安装管理。由于组织新的发行版时并没有一个统一的标准，所以不同厂商的发行版各有特点也各有不足。

Linux 内核的终极的来源虽然只有一个，但是可以为其改进和发展作出贡献的志愿者人数却并无限制。同时，考虑到一些特殊的应用，一些开发商或机构往往对内核加以修改和补充，形成一些针对特殊环境或要求的变种。例如，针对"嵌入式"系统的要求，有人就开发出 Embedded Linux；针对有"硬实时"要求的系统，有人就开发了 RT-Linux；针对手持式计算机的要求，有人就开发出了 Baby Linux；等等。当然，中文 Linux 也是其中的一类。每当有新的 Linux 内核版本发布时，这些变种版本通常也很快就会推出相应的新版本。根据 FSF 对自由软件版权的规定（GPL），这些变种版本对内核的修改与补充必须公开源代码。

许多人以为，既然 Linux 是免费的公开软件，那就无所谓版权的问题了。其实不然。Linux 以及 Linux 内核源代码，是有版权保护的，只不过这版权归公众（或者说全人类）所有，由自由软件基金会 FSF 管理。FSF 为所有的 GNU 软件制定了一个公用许可证制度，称为 GPL（General Public License），也叫 Copyleft，这是与通常所讲的版权即 Copyright 截然不同的制度。Copyright 即通常意义下的版权，保护作者对其作品及其衍生品的独占权，而 Copyleft 则允许用户对作品进行复制、修改，但要求用户承担 GPL 规定的一些义务。按 GPL 规定，允许任何人免费地使用 GNU 软件，并且可以用 GNU 软件的源代码重构可执行代码。进一步，GPL 还允许任何人免费地取得 GNU 软件及其源代码，并且再加以发布甚至出售，但必须要符合 GPL 的某些条款。简而言之，这些条款规定 GNU 软件以及在 GNU 软件的基础加以修改而成的软件，在发布（或转让、出售）时必须要申明该软件出自 GNU（或者源自 GNU），并且必须要保证让接收者能够共享源代码，能从源代码重构可执行代码。换言之，如果一个软件是在 GNU 源代码的基础加以修改、扩充而来的，那么这个软件的源代码就也必须对使用者公开（注意产品的出售与源代码的公开并不一定相矛盾)。通过这样的途径，自由软件的阵容就会像滚雪球一样越滚越大。不过，如果一个软件只是通过某个 GNU 软件的用户界面（API）使用该软件，则不受 GPL 条款的约束或限制。总之，GPL 的主要目标是：使自由软件及其衍生产品继续保持开放状态，从整体上促进软件的共享和重复使用。具体到 Linux 的内核来说，如果你对内核源代码的某些部分作了修改，或者在你的程序中引用了 Linux 内核中的某些段落，你就必须加以申明并且公开你的源代码。但是，如果你开发了一个用户程序，只是通过系统调用的界面使用内核，则你自己拥有完全的知识产权，不受 GPL 条款的限制。

应该说，FSF 的构思是很巧妙也是很合理的，其目的也是很高尚的。

说到高尚，此处顺便多说几句。美国曾经出过两本很有些影响的书，一本叫 *Undocumented DOS*，另一本叫 *Undocumented Windows*，两本书均被列入 DOS/Windows 系统程序员的必备工具书。在这两本书中，作者们（Andrew Schulman，David Maxey 以及 Matt Pietrek 等）一一列举了经过他们辛勤努力才破译和总结出来的 DOS/Windows API（应用程序设计界面）实际上提供了但却没有列入 Microsoft 技术资料的许多有用（而且重要）的功能。作者们认为，Microsoft 没有将这些功能收入其技术资料的原因是无法用疏忽或遗漏加以解释的，而只能是故意向用户隐瞒。Microsoft 既是操作系统的提供者，同时又是一个应用程序的开发商，通过向其它的应用程序开发商隐瞒一些操作系统界面上的技术关键，就使那些开发商无法与 Microsoft 公平竞争，从而使 Microsoft 可以通过对关键技术的垄断达到对 DOS/Windows 应用软件市场的垄断。作者们在书中指责 Microsoft 这样做不仅有道德上的问题，也有法律上的问题。是否涉及法律问题姑且不论，书中所列的功能确实都是存在的，可以通过实验证实，也

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

确实没有写入 Microsoft 向客户提供的技术资料。

要是将 FSF 与 Microsoft 放在一起，则二者恰好成为鲜明的对比。差别之大，读者不难做出自己的结论。

GPL 的正文包含在一个叫 COPYING 的文件中。在通过光盘安装的 Linux 系统中，该文件的路径名为/usr/scr/linux/COPYING。而在下载的 Linux 内核 tar 文件中，经过解压后该文件在顶层目录中。有兴趣或有需要的读者可以（而且应该）仔细阅读。

1.2 Intel X86 CPU 系列的寻址方式

Intel 可以说是资格最老的微处理器芯片制造商了，历史上的第一个微处理芯片 4004 就是 Intel 制造的。所谓 X86 系列，是指 Intel 从 16 位微处理器 8086 开始的整个 CPU 芯片系列，系列中的每种型号都保持与以前的各种型号兼容，主要有 8086、8088、80186、80286、80386、80486 以及以后各种型号的 Pentium 芯片。自从 IBM 选择 8088 用于 PC 个人计算机以后，X86 系列的发展就与 IBM PC 及其兼容机的发展休戚相关了。其中 80186 并不广为人知，就与 IBM 当初决定停止在 PC 机中使用 80186 有关。限于篇幅，本书不对这个系列的系统结构作全面的介绍，而只是结合 Linux 内核的存储管理对其寻址方式作一些简要的说明。

在 X86 系列中，8086 和 8088 是 16 位处理器，而从 80386 开始为 32 位处理器，80286 则是该系列从 8088 到 80386，也就是从 16 位到 32 位过渡时的一个中间步骤。80286 虽然仍是 16 位处理器，但是在寻址方式上开始了从"实地址模式"到"保护模式"的过渡。

当我们说一个 CPU 是"16 位"或"32 位"时，指的是处理器中"算术逻辑单元"（ALU）的宽度。系统总线中的数据线部分，称为"数据总线"，通常与 ALU 具有相同的宽度（但有例外）。那么"地址总线"的宽度呢？最自然的地址总线宽度是与数据总线一致。这是因为从程序设计的角度来说，一个地址，也就是一个指针，最好是与一个整数的长度一致。但是，如果从 8 位 CPU 寻址能力的角度来考虑，则这实际上是不现实的，因为一个 8 位的地址只能用来寻访 256 个不同的地址单元，这显然太小了。所以，一般 8 位 CPU 的地址总线都是 16 位的。这也造成了一些 8 位 CPU 在内部结构上的一些不均匀性，在 8 位 CPU 的指令系统中常常会发现一些实际上是 16 位的操作。当 CPU 的技术从 8 位发展到 16 位的时候，本来地址总线的宽度是可以跟数据总线一致了，但是当时人们已经觉得由 16 位地址所决定的地址空间（64K）还是太小，还应该加大。加到多大呢？结合当时人们所能看到的微型机的应用前景，以及存储器芯片的价格，Intel 决定采用 1M，也就是说 64K 的 16 倍，那时觉得应该是足够了。确实，1M 字节的内存空间在当时已经很使一些程序员激动不已了，那时候配置齐全的小型机，甚至大型机也只不过是 4M 字节的内存空间。在计算机的发展史上，几乎每一个技术决策，往往很快就被事后出现的事实证明是估计不足的。

既然 Intel 决定了在其 16 位 CPU，即 8086 中采用 1M 字节的内存地址空间，地址总线的宽度也就相应地确定了，那就是 20 位。这样，一个问题就摆在了 Intel 的设计人员面前：虽然地址总线的宽度是 20 位，但 CPU 中 ALU 的宽度却只有 16 位，也就是说可直接加以运算的指针的长度是 16 位的。如何来填补这个空隙呢？可能的解决方案当然有很多种。例如，可以像在一些 8 位 CPU 中那样，增设一些 20 位的指令专用于地址运算和操作，但是那样又会造成 CPU 内部结构的不均匀性。再如，当时的

PDP-11 小型机也是 16 位的，但是结合其 MMU（内存管理单元）可以将 16 位的地址映射到 24 位的地址空间。结果，Intel 设计了一种在当时看来还不失巧妙的方法，即分段的方法。

Intel 在 8086 CPU 中设置了四个“段寄存器”：CS、DS、SS 和 ES，分别用于可执行代码即指令、数据、堆栈和其它。每个段寄存器都是 16 位的，对应于地址总线中的高 16 位。每条“访内”指令中的“内部地址”都是 16 位的，但是在送上地址总线之前都在 CPU 内部自动地与某个段寄存器中的内容相加，形成一个 20 位的实际地址。这样，就实现了从 16 位内部地址到 20 位实际地址的转换，或者“映射”。这里要注意段寄存器中的内容对应于 20 位地址总线中的高 16 位，所以在相加时实际上是拿内部地址中的高 12 位与段寄存器中的 16 位相加，而内部地址中的低 4 位保留不变。这个方法与操作系统理论中的“段式内存管理”相似，但并不完全一样，主要是没有地址空间的保护机制。对于每一个由段寄存器的内容确定的“基地址”，一个进程总是能够访问从此开始的 64K 字节的连续地址空间，而无法加以限制。同时，可以用来改变段寄存器内容的指令也不是什么“特权指令”，也就是说，通过改变段寄存器的内容，一个进程可以随心所欲地访问内存中的任何一个单元，而丝毫不受到限制。不能对一个进程的内存访问加以限制，也就谈不上对其它进程以及系统本身的保护。与此相应，一个 CPU 如果缺乏对内存访问的限制，或者说保护，就谈不上什么内存管理，也就谈不上是现代意义上的中央处理器。由于 8086 的这种内存寻址方式缺乏对内存空间的保护，所以为了区别于后来出现的“保护模式”，就称为“实地址模式”。

显然，在实地址模式上是无法建造起现代意义上的“操作系统”的。

针对 8086 的这种缺陷，Intel 从 80286 开始实现其“保护模式”（Protected Mode，但是早期的 80286 只能从实地址模式转入保护模式，却不能从保护模式转回实地址模式）。同时，不久以后 32 位的 80386 CPU 也开发成功了。这样，从 8088/8086 到 80386 就完成了一次从比较原始的 16 位 CPU 到现代的 32 位 CPU 的飞跃，而 80286 则变成这次飞跃的一个中间步骤。从 80386 以后，Intel 的 CPU 历经 80486、Pentium、Pentium II 等等型号，虽然在速度上提高了好几个量级，功能上也有了不小的改进，但基本上属于同一种系统结构中的改进与加强，而并无重大的质的改变，所以统称为 i386 结构，或 i386 CPU。下面我们将以 80386 为背景，介绍 i386 系列的保护模式。

80386 是个 32 位 CPU，也就是说它的 ALU 数据总线是 32 位的。我们在前面说过，最自然的地址总线宽度是与数据总线一致。当地址总线的宽度达到 32 位时，其寻址能力达到了 4G（4 千兆），对于内存来说似乎是足够了。所以，如果新设计一个 32 位 CPU 的话，其结构应该是可以做到很简洁、很自然的。但是，80386 却无法做到这一点。作为一个产品系列中的一员，80386 必须维持那些段寄存器，还必须支持实地址模式，与此同时又要能支持保护模式。而保护模式是完全另搞一套，还是建立在段寄存器的基础上以保持风格上的一致，并且还能节约 CPU 的内部资源呢？这对于 Intel 的设计人员无疑又是一次挑战。

Intel 选择了在段寄存器的基础上构筑保护模式的构思，并且保留段寄存器为 16 位（这样才可以利用原有的四个段寄存器），但是却又增添了两个段寄存器 FS 和 GS。为了实现保护模式，光是用段寄存器来确定一个基地址是不够的，至少还得要有一个地址段的长度，并且还需要一些其它的信息，如访问权限之类。所以，这里需要的是一个数据结构，而并非一个单纯的基地址。对此，Intel 设计人员的基本思路是：在保护模式下改变段寄存器的功能，使其从一个单纯的基地址（变相的基地址）变成指向这样一个数据结构的指针。这样，当一条访内存指令发出一个内存地址时，CPU 就可以这样来归纳出实际上应该放上数据总线的地址：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

(1) 根据指令的性质来确定应该使用哪一个段寄存器，例如转移指令中的地址在代码段，而取数指令中的地址在数据段。这一点与实地址模式相同。
(2) 根据段寄存器的内容，找到相应的“地址段描述结构”。
(3) 从地址段描述结构中得到基地址。
(4) 将指令中发出的地址作为位移，与段描述结构中规定的段长度相比，看看是否越界。
(5) 根据指令的性质和段描述符中的访问权限来确定是否越权。
(6) 将指令中发出的地址作为位移，与基地相加而得出实际的“物理地址”。

虽然段描述结构存储在内存中，在实际使用时却将其装载入 CPU 中的一组“影子”结构，而 CPU 在运行时则使用其在 CPU 中的“影子”。从“保护”的角度考虑，在由（指令给出的）内部地址（或者说“逻辑地址”）转换成物理地址的过程中，必须要在某个环节上对访问权限进行比对，以防止不具备特权的用户程序通过玩弄某些诡计（例如修改段寄存器的内容，修改段描述结构的内容等），得以非法访问其它进程的空间或系统空间。

明白了这个思路，80386 的段式内存管理机制就比较容易理解了（还是很复杂）。下面就是此机制的实际实现。

首先，在 80386 CPU 中增设了两个寄存器：一个是全局性的段描述表寄存器 GDTR（global descriptor table register），另一个是局部性的段描述表寄存器 LDTR（local descriptor table register)，分别可以用来指向存储在内存中的一个段描述结构数组，或者称为段描述表。由于这两个寄存器是新增设的，不存在与原有的指令是否兼容的问题，访问这两个寄存器的专用指令便设计成“特权指令”。

在此基础上，段寄存器的高 13 位（低 3 位另作它用）用作访问段描述表中具体描述结构的下标（index），如图 1.1 所示。

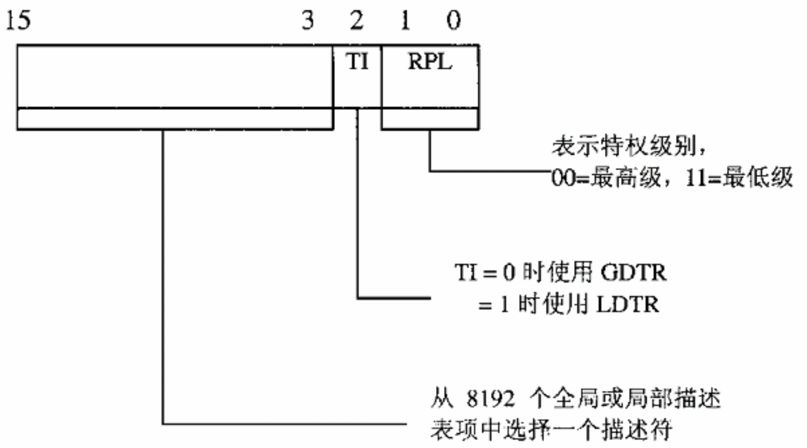


图 1.1　段寄存器定义

GDTR 或 LDTR 中的段描述表指针和段寄存器中给出的下标结合在一起，才决定了具体的段描述表项在内存中的什么地方，也可以理解成，将段寄存器内容的低 3 位屏蔽掉以后与 GDTR 或 LDTR 中的基地址相加得到描述表项的起始地址。因此就无法通过修改描述表项的内容来玩弄诡计，从而起到保护的作用。每个段描述表项的大小是 8 个字节，每个描述表项含有段的基地址和段的大小，再加上其它一些信息，其结构如图 1.2 所示。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

<table>
<tr><td colspan="4">B31—B24</td><td>G</td><td>D</td><td>A</td><td></td><td>L19—L16</td></tr>
<tr><td>P</td><td>DPL</td><td>S</td><td>type</td><td colspan="5">B23—B16</td></tr>
<tr><td colspan="9">B15—B0</td></tr>
<tr><td colspan="9">L15—L0</td></tr>
</table>

图 1.2　8 字节段描述符表项的定义

结构中的 B31～B24 和 B23～B16 分别为基地址的 bit16～bit23 和 bit24～bit31。而 L19～L16 和 L15～L0 则为段长度（Limit）的 bit0～bit15 和 bit16～bit19。其中 DPL 是个 2 位的位段，而 type 是一个 4 位的位段。它们所在的整个字节分解如图 1.3 所示。

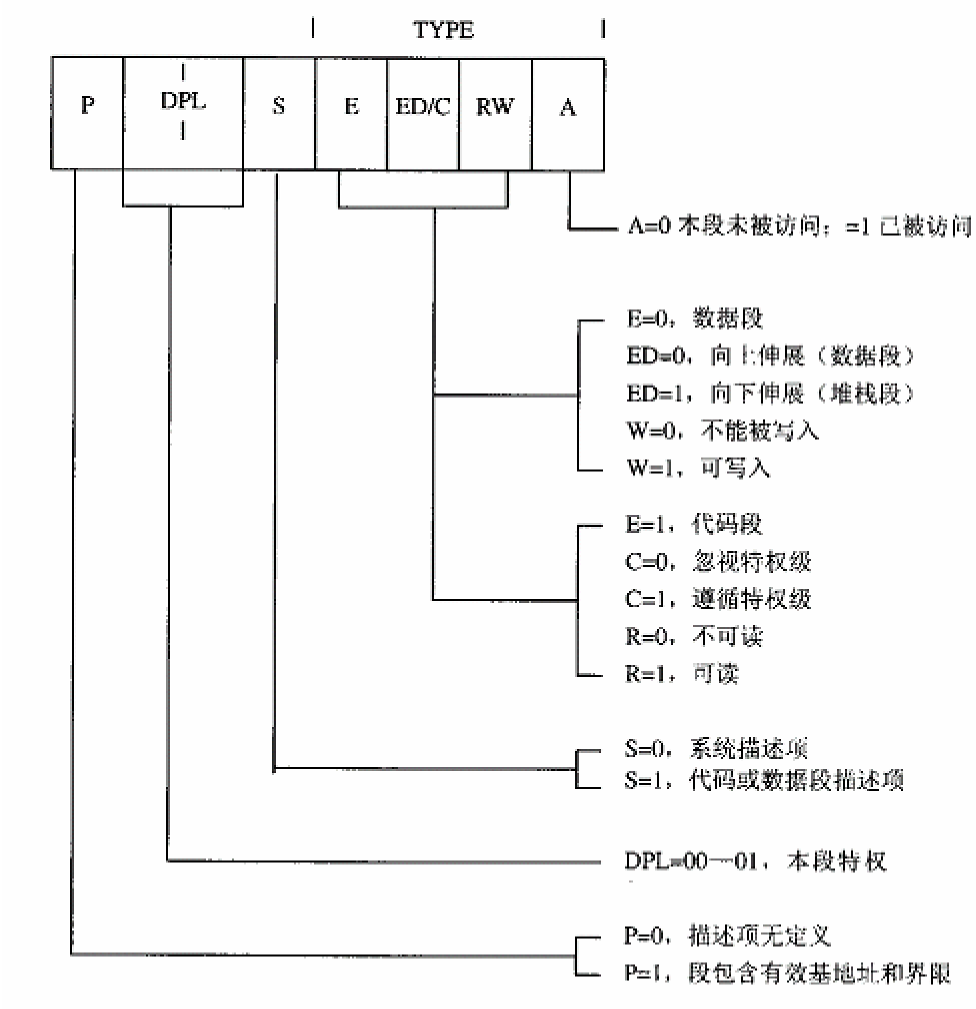


图 1.3　段描述表项 TYPE 字节的定义

我们也可以用一段“伪代码”来说明整个段描述结构：

```
typedef struct {
    unsigned  int  base24_31:8;    /* 基地址的最高 8 位 */
    unsigned  int  g:1;    /* granularity，表段的长度单位，0 表示字节，1 表示 4KB */
    unsigned  int  d_b:1;  /* default operation size 存取方式，0=16 位，1=32 位 */
    unsigned  int  unused:1;          /* 固定设置成 0 */
    unsigned  int  avl:1;            /* avalaible，可供系统软件使用 */
    unsigned  int  seg_limit_16_19:4;  /* 段长度的最高 4 位 */
    unsigned  int  p:1;    /* segment present，为 0 时表示该段的内容不在内存中 */
    unsigned  int  dpl:2;  /* Descriptor privilege level，访问本段所需权限 */
    unsigned  int  s:1;              /* 描述项类型，1 表示系统，0 表示代码或数据 */
    unsigned  int  type:4;           /* 段的类型，与上面的 S 标志位一起使用 */
    unsigned  int  base_0_23:24;       /* 基地址的低 24 位 */
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
    unsigned  int  seg_limit_0_15:16;  /* 段长度的低 16 位 */
} 段描述项;
```

以这里的位段 type 为例，“ :4”表示其宽度为 4 位。整个数据结构的大小为 64 位，即 8 个字节。

读者一定会问：为什么把段描述项定义成这样一种奇怪的结构？例如，为什么基地址的高 8 位和低 24 位不连在一起？最自然也最合理的解释就是：开始时 Intel 的意图是 24 位地址空间，后来又改成 32 位地址空间。这也可以从段长度字段也是拆成两节得到印证：当 g 标志位为 1 时，长度的单位为 4KB，而段长度字段的低 16 位的容量是 64K，所以一个段的最大可能长度为 64K×4K=256M，而这正是 24 位地址空间的大小。所以，可以看出，Intel 起先意欲使用 24 位地址空间，不久又认识到应该用 32 位，但是 80286 已经发售出去了，于是就只好修修补补。当时的 Intel 确实给人一种“小脚女人走路”的感觉。

每当一个段寄存器的内容改变时（通过 MOV、POP 等指令或发生中断等事件），CPU 就把由这段寄存器的新内容所决定的段描述项装入 CPU 内部的一个“影子”描述项。这样，CPU 中有几个段寄存器就有几个影子描述项，所以也可以看作是对段寄存器的扩充。扩充后的段寄存器分成两部分，一部分是可见的（对程序而言），还与原先的段寄存器一样；另一部分是不可见的，就是用来存放影子描述项的空间，这一部分是专供 CPU 内部使用的。

在 80386 的段式内存管理的基础上，如果把每个段寄存器都指向同一个描述项，而在该描述项中则将基地址设成 0，并将段长度设成最大，这样便形成一个从 0 开始覆盖整个 32 位地址空间的一个整段。由于基地址为 0，此时的物理地址与逻辑地址相同，CPU 放到地址总线上去的地址就是在指令中给出的地址。这样的地址有别于由“段寄存器 / 位移量”构成的“层次式”地址，所以 Intel 称其为“平面（Flat）”地址。Linux 内核的源代码（更确切地应该说是 gcc）采用平面地址。这里要指出，平面地址的使用并不意味着绕过了段描述表、段寄存器这—整套段式内存管理的机制，而只是段式内存管理的一种使用特例。

关于 80386 的段式内存管理就先介绍这些，以后随着代码分析的进展视需要再加以补充。读者想要了解完整的细节可以参阅 Intel 的有关技术资料。

利用 80386 对段式内存管理的硬件支持，可以实现段式虚存管理。如前所述，当一个段寄存器的内容改变时，CPU 要根据新的段寄存器内容以及 GDTR 或 LDTR 的内容找到相应的段描述项并将其装入 CPU 中。在此过程中，CPU 会检查该描述项中的 p 标志位（表示“ present”），如果 p 标志位为 0，就表示该描述项所指向的那一段内容不在内存中（也就是说，在磁盘上的某个地方），此时 CPU 会产生一次异常（exception，类似于中断），而相应的服务程序便可以从磁盘交换区将这一段的内容读入内存中的某个地方，并据此设置描述项中的基地址，再将 p 标志位设置成 1。相应地，内存中暂时不用的存储段则可以写入磁盘，并将其描述项中 p 标志位改成 0。

对段式内存管理的支持只是 i386 保护模式的一个组成部分。如果没有系统状态和用户状态的分离，以及特权指令（只允许在系统状态下使用）的设立，那么尽管有了前述的段式内存管理，也还不能起到保护的效果。前面已经提到过特权指令的设置，如用来装入和存储 GDTR 和 LDTR 的指令 LGDT/LLDT 和 SGDT/SLDT 等就都是特权指令。正是由于这些指令都只能在系统状态（也就是在操作系统的内核中）使用，才使得用户程序不但不能改变 GDTR 和 LDTR 的内容，还因为既无法确知其段描述表在内存中的位置，又无法访问其段描述表所在的空间（只能在系统状态下才能访问），从而无法通过修改段描述项来打破系统的保护机制。那么，80386 怎么来分隔系统状态和用户状态，并且提供在

PDF-XCHANGE Click to buy NOW! www.docu-track.com

两种状态之间切换的机制呢？

80386 并不只是像一般 CPU 通常所做的那样，划分出系统状态和用户状态，而是划分成四个特权级别，其中 0 级为最高，3 级为最低。每一条指令也都有其适用级别，如前述的 LGDT，就只有在 0 级的状态下才能使用，而一般的输入 / 输出指令（IN，OUT）则规定为 0 级或 1 级。通常，用户的应用程序都是 3 级。一般程序的当前运行级别由其代码段的局部描述项（即由段寄存器 CS 所指向的局部段描述项）中的 dpl 字段决定（dpl 表示 " descriptor privilege level）。当然，每个描述项中的 dpl 字段都是在 0 级状态下由内核设定的。而全局段描述的 dpl 字段，则又有所不同，它是表示所需的级别。

前面讲过，16 位的段寄存器中的高 13 位用作下标来访问段描述表，而低 3 位是干什么的呢？我们还是通过一段伪代码来说明：

```
typedef struct {
    unsigned  short seg_idx:13;  /* 13 位的段描述项下标 */
    unsigned  short ti:1;         /* 段描述表指示位，0 表示 GDT，1 表示 LDT */
    unsigned  short rpl:2;       /* Requested Privilege Level，要求的优先级别 */
} 段寄存器;
```

当段寄存器 CS 中的 ti 位为 1 时，表示要使用全局段描述表，为 0 时，则表示要使用局部段描述表。而 rpl 则表示所要求的权限。当改变一个段寄存器的内容时，CPU 会加以检查，以确保该段程序的当前执行权限和段寄存器所指定要求的权限均不低于所要访问的那一段内存的权限 dpl。

至于怎样在不同的执行权限之间切换，我们将在进程调度、系统调用和中断处理的有关章节中讨论。此外，除了全局段描述表指针 GDTR 和局部段描述表指针 LDTR 两个寄存器外，其实 i386 CPU 中还有个中断向量表指针寄存器 IDTR、与进程（在 Intel 术语中称为“任务”，Task）有关的寄存器 TR 以及描述任务状态的“任务状态段 ”TSS 等，这些都将在其它章节中有需要时再加以介绍。Intel 在实现 i386 的保护模式时将 CPU 的执行状态分成四级，意图是为满足更为复杂的操作系统和运行环境的需要。有些操作系统，如 OS/2 中，也确实用了。但是很多人都怀疑是否真有必要搞得那么复杂。事实上，几乎所有广泛使用的 CPU 都没有这么复杂。而且，在 80386 上实现的各种 Unix 版本，包括 Linux，都只用了两个级别，即 0 级和 3 级，作为系统状态和用户状态。本书在以后的讨论中将沿用 Unix 的传统称之为系统状态和用户状态。

1.3　i386 的页式内存管理机制

学过操作系统原理的读者都知道，内存管理有两种，一种是段式管理，另一种是页式管理，而页式管理更为先进。从 80 年代中期开始，页式内存管理进入了各种操作系统（以 Unix 为主）的内核，一时成为操作系统领域的一个热点。

Intel 从 80286 开始实现其“保护模式”，也即段式内存管理。但是很快就发现，光有段式内存管理而没有页式内存管理是不够的，那样会使它的 X86 系列逐渐失去竞争力以及作为主流 CPU 产品的地位。因此，在不久以后的 80386 中就实现了对页式内存管理的支持。也就是说，80386 除了完成并完善从 80286 开始的段式内存管理的同时还实现了页式内存管理。

前面讲过，80386 的段式内存管理机制，是将指令中结合段寄存器使用的 32 位逻辑地址映射（转

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

换）成同样是 32 位的物理地址。之所以称为“物理地址”，是因为这是真正放到地址总线上去，并用以寻访物理上存在着的具体内存单元的地址。但是，段式存储管理机制的灵活性和效率都比较差。一方面“段”是可变长度的，这就给盘区交换操作带来了不便；另一方面，如果为了增加灵活性而将一个进程的空间划分成很多小段时，就势必要求在程序中频繁地改变段寄存器的内容。同时，如果将段分小，虽然一个段描述表中可以容纳 8192 个描述项（因为有 13 位下标），也未必就能保证足够使用。所以，比较好的办法还是采用页式存储管理。本来，页式存储管理并不需要建立在段式存储管理的基础之上，这是两种不同的机制。可是，在 80386 中，保护模式的实现是与段式存储密不可分的。例如，CPU 的当前执行权限就是在有关的代码段描述项中规定的。读过 Unix 早期版本的读者不妨将此与 PDP-11 中的情况作一对比。在 PDP-11 中 CPU 的当前执行权限存放在一个独立的寄存器 PSW 中，而与任何其它的数据结构没有关系。因此，在 80386 中，既然决定利用部分已经存在的资源，而不是完全另起炉灶，那就无法绕过段式存管来实现页式存管。也就是说，80386 的系统结构决定了它的页式存管只能建立在段式存管的基础上。这也意味着，页式存管的作用是在由段式存管所映射而成的地址上再加上一层地址映射。由于此时由段式存管映射而成的地址不再是“物理地址”了，Intel 就称之为“线性地址”。于是，段式存管先将逻辑地址映射成线性地址，然后再由页式存管将线性地址映射成物理地址；或者，当不使用页式存管时，就将线性地址直接用作物理地址。

80386 把线性地址空间划分成 4K 字节的页面，每个页面可以被映射至物理存储空间中任意一块 4K 字节大小的区间（边界必须与 4K 字节对齐）。在段式存管中，连续的逻辑地址经过映射后在线性地址空间还是连续的。但是在页式存管中，连续的线性地址经过映射后在物理空间却不一定连续（其灵活性也正在于此)。这里需要指出的是，虽然页式存管是建立在段式存管的基础上，但一旦启用了页式存管，所有的线性地址都要经过页式映射，连 GDTR 与 LDTR 中给出的段描述表起始地址也不例外。

由于页式存管的引入，对 32 位的线性地址有了新的解释（以前就是物理地址）：

```
typedef struct {
    unsigned  int  dir:10;     /* 用作页面表目录中的下标，该目录项指向一个页面表 */
    unsigned  int  page:10;    /* 用作具体页面表中的下标，该表项指向一个物理页面 */
    unsigned  int  offset:12;  /* 在 4K 字节物理页面内的偏移量 */
} 线性地址;
```

这个结构可以用图 1.4 形象地表示。

31　　　　　　22	21　　　　　　12	11　　　　　　0
页面目录 dir	页面表	页内偏移 offset

图 1.4　线性地址的格式

可以看出，在页面目录中共有 $2^{10} = 1024$ 个目录项，每个目录项指向一个页面表，而在每个页面表中又共有 1024 个页面描述项。类似于 GDTR 和 LDTR，又增加了一个新的寄存器 CR3 作为指向当前页面目录的指针。这样，从线性地址到物理地址的映射过程为：

(1)　从 CR3 取得页面目录的基地址。
(2)　以线性地址中的 dir 位段为下标，在目录中取得相应页面表的基地址。
(3)　以线性地址中的 page 位段为下标，在所得到的页面表中取得相应的页面描述项。
(4)　将页面描述项中给出的页面基地址与线性地址中的 offset 位段相加得到物理地址。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

(5) 上述映射过程可用图 1.5 直观地表示。

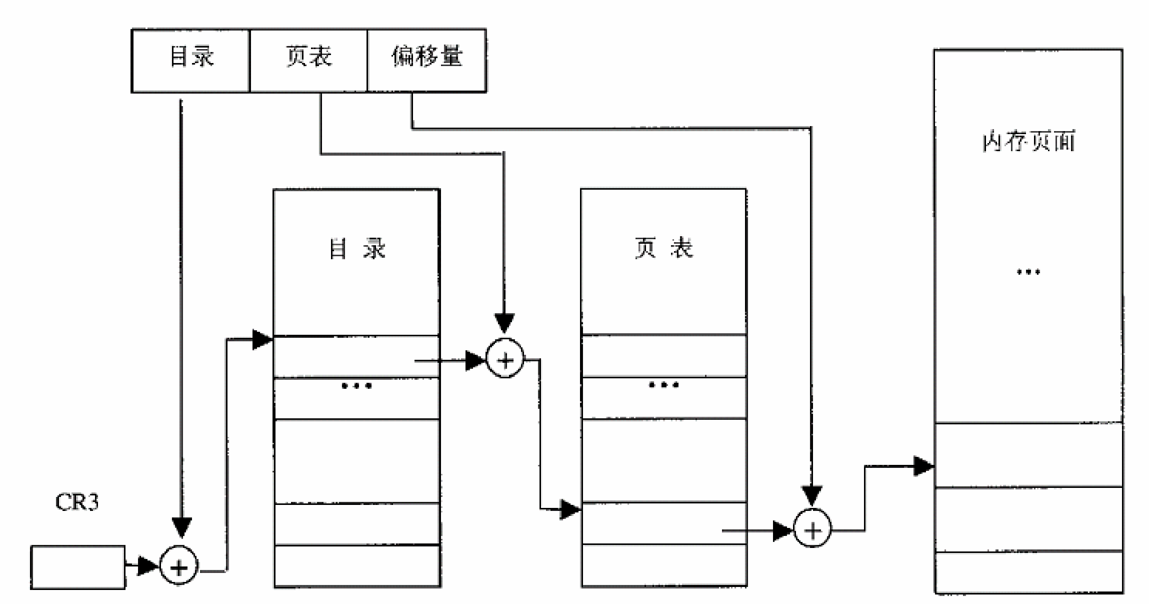


图 1.5 页式映射示意图

那么，为什么要使用两个层次，先找到目录项，再找到页面描述项，而不是像在使用段寄存器时那样一步到位呢？这是出于空间效率的考虑。如果将线性地址中的 dir 和 page 两个位段合并在一起是 20 位，因此页面表的大小就将是 1K×1K=1M 个表项。由于每个页面的大小为 4K 字节，总的空间大小仍为 4K×1M=4G，正好是 32 位地址空间的大小。但是，实际上很难想象有一个进程会需要用到 4G 的全部空间，所以大部分表项势必是空着的。可是，在一个数组中，即使是空着不用的表项也占用空间，这样就造成了浪费。而若分成两层，则页表可以视需要而设置，如果目录中某项为空，就不必设立相应的页表，从而省下了存储空间。当然，在最坏的情况下，如果一个进程真的要用到全部 4G 的存储空间，那就不仅不能节省，反而要多消耗一个目录所占用的空间，但那概率基本上是 0。另外，一个页面的大小是 4K 字节，而每一个页面表项或目录表项的大小是 4 个字节。1024 个表项正好也是 4K 字节，恰好可以放在一个页面中。而若多于 1024 项就要使目录或页面表跨页面存放了。也正为此，在 64 位的 Alpha CPU 中页面的大小是 8K 字节，因为目录表项和页面表项的大小都变成了 8 个字节。

如前所述，目录项中含有指向一个页面表的指针，而页面表项中则含有指向一个页面起始地址的指针。由于页面表和页面的起始地址都总是在 4K 字节的边界上，这些指针的低 12 位都永远是 0。这样，在目录项和页表项中都只要有 20 位用于指针就够了，而余下的 12 位则可以用于控制或其它的目的。于是，目录项的结构为：

```
typedef struct {
    unsigned  int  ptba:20;       /* 页表基地址的高 20 位 */
    unsigned  int  avail:3;       /* 供系统程序员使用 */
    unsigned  int  g:1;          /* global，全局性页面 */
    unsigned  int  ps:1;          /* 页面大小，0 表示 4K 字节 */
    unsigned  int  reserved:1;    /* 保留，永远是 0 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    unsigned  int  a:1;          /* accessed，已被访问过 */
    unsigned  int  pcd:1;         /* 关闭（不使用）缓冲存储器 */
    unsigned  int  pwt:1;         /* Write Through，用于缓冲存储器 */
    unsigned  int  u_s:1;         /* 为 0 时表示系统（或超级）权限，为 1 时表示用户权限 */
    unsigned  int  r_w:1;         /* 只读或可写 */
    unsigned  int  p:1;          /* 为 0 时表示相应的页面不在内存中 */
} 目录项;
```

目录项的直观表示如图 1.6 表示。

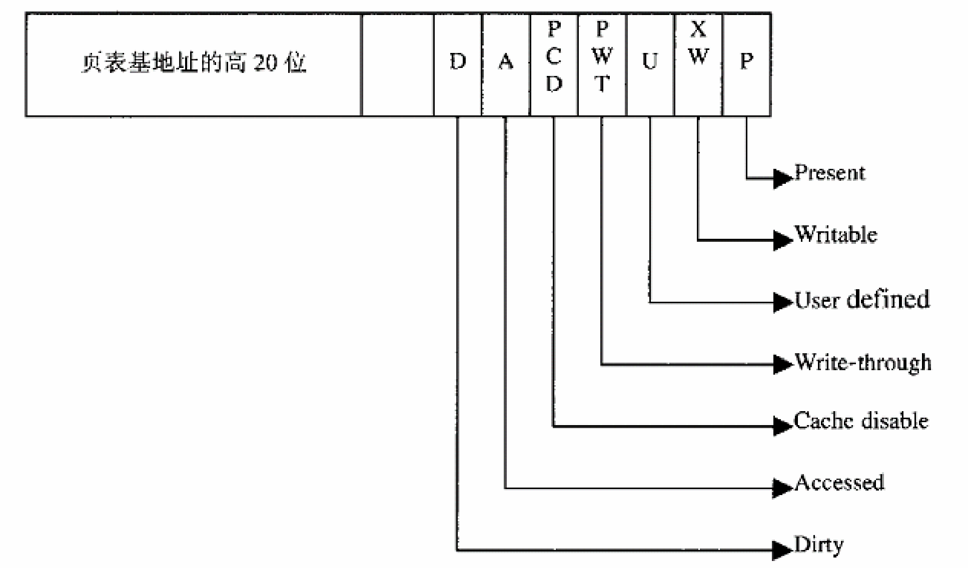


图 1.6　页目录项示意图

页表项的结构基本上与此相同，但没有“页面大小”位 ps，所以第 8 位保留不用，但第 7 位（在目录项中保留不用）则为 D（Dirty）标志，表示该页面已经被写过，所以已经“脏”了。当页面表项或目录项中的最低位 p 为 0 时，表示相应的页面或页面表不在内存，根据其它一些有关寄存器的设置，CPU 可以产生一个“页面错”（Page Fault）异常（也称为缺页中断，但异常和中断其实是有区别的）。这样，内核中的有关异常服务程序就可以从磁盘上的页面交换区将相应的页面读入内存，并且相应地设置表项中的基地址，并将 p 位设置成 1。相反，也可以将内存中暂不使用的页面写入磁盘的交换区，然后将相应页面表项的 p 位设置为 0。这样，就可以实现页式虚存了。当 p 位为 0 时，表项的其余各位均无意义，所以可被用来临时存储其它信息，如被换出的页面在磁盘上的位置等等。

当目录项中的 ps（page size）位为 0 时，包含在由该目录项所指的页面表中所有页面的大小都是 4K 字节，这也是目前在 Linux 内核中所采用的页面大小。但是，从 Pentium 处理器开始，Intel 引入了 PSE 页面大小扩充机制。当 ps 位为 1 时，页面的大小就成了 4M 字节，而页面表就不再使用了。这时候，线性地址中的低 22 位就全部用作在 4M 字节页面中的位移。这样，总的寻址能力还是没有改变，即 1024×4M=4G，但是映射的过程减少了一个层次。随着内存容量和磁盘容量的日益增加，磁盘访问速度的显著提高，以及对图像处理要求的日益增加，4M 字节的页面大小有可能会成为主流。在这一点上，Intel 倒还是有远见的。

最后，i386 CPU 中还有个寄存器 CR0，其最高位 PG 是页式映射机制的总开关。当 PG 位被设置成 1 时，CPU 就开启了页式存储管理的映射机制。

从 Pentium Pro 开始，Intel 又作了扩充。这一次扩充的是物理地址的宽度。Intel 在另一个控制寄存器 CR4 中又增加了一位 PAE（表示 Physical Address Extension），当 PAE 位设置成 1 时，地址总线的宽度就变成了 36 位（又增加了 4 位）。与此相应，页式存储管理的映射机制也自然地有所改变。不过大多数用户都还不需要使用 36 位（64G）物理地址空间，所以这里从略，有兴趣的读者可以参阅 Intel 的有关技术资料或专著。此外，Intel 已经推出了 64 位的 IA-64 系统结构，Linux 内核也已经支持 IA-64 系统结构。事实上，Linux 原来就已经在 Alpha CPU 上支持 64 位地址。除存储管理外，80386 还有很强的高速缓冲存储和流水线功能。但是对于软件、对于操作系统的内核来说，那在很大程度上是透明的，所以本书将仅在有必要时才加以简单的说明，而不在此详述了。

1.4 Linux 内核源代码中的 C 语言代码

Linux 内核的主体是以 GNU 的 C 语言编写的，GNU 为此提供了编译工具 gcc。GNU 对 C 语言本身（在 ANSI C 基础上）作了不少扩充，可能是读者尚未见到过的。另一方面，由于是内核代码，往往会用到一些在应用程序设计中不常见的语言成分或编程技巧，也许使读者感到陌生。本书并非介绍 GNU C 语言的专著，也非技术手册，所以不在这里一一列举和详细讨论这些扩充和技巧。再说，离开具体的情景和上下文，罗列一大堆规则，对于读者恐怕也没有多大帮助。所以，我们在这里只对可能会影响读者阅读 Linux 内核源程序，或使读者感到困惑的一些扩充和技巧先作一些简单的介绍。以后，随着具体的情景和代码的展开，在需要时还会结合实际加以补充。

首先，gcc 从 C++语言中吸收了“inline”和“const”。其实，GNU 的 C 和 C++是合为一体的，gcc 既是 C 编译又是 C++编译，所以从 C++中吸收一些东西到 C 中是很自然的。从功能上说，inline 函数的使用与#define 宏定义相似，但更有相对的独立性，也更安全。使用 inline 函数也有利于程序调试。如果编译时不加优化，则这些 inline 函数就是普通的、独立的函数，更便于调试。调试好了以后，再采用优化重新编译一次，这些 inline 函数就像宏操作一样融入了引用处的代码中，有利于提高运行效率。由于 inline 函数的大量使用，相当一部分的代码从.c 文件移入了.h 文件中。

还有，为了支持 64 位的 CPU 结构（Alpha 就是 64 位的），gcc 增加了一种新的基本数据类型“long long int”，该类型在内核代码中常常用到。

许多 C 语言都支持一些“属性描述符”（attribute），如“aligned”、“packed”等等；gcc 也支持不少这样的描述符。这些描述符的使用等于是在 C 语言中增加了一些新的保留字。可是，在原来的 C 语言（如 ANSI C）中这些词并非保留字，这样就有可能产生一些冲突。例如，gcc 支持保留字 inline，可是由于“inline”原非保留字（在 C++中是保留字），所以在老的代码中可能已经有一变量名为 inline，这样就产生了冲突。为了解决这个问题，gcc 允许在作为保留字使用的“inline”前、后都加上“__”，因而“__inline__”等价于保留字“inline”。同样的道理，“__asm__”等价于“asm”。这就是我们在代码中有时候看到“asm”，而有时候又看到“__asm__”的原因。

gcc 还支持一个保留字“attribute”，用来作属性描述。如：

```
struct foo {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        char    a;
        int     x[z]    attribute__ ((packed));
    }
```

这里属性描述“ packed”表示在字符 a 与整型数组 x 之间不应为了与 32 位长整数边界对齐而留下空洞。这样，“ packed”就不会与变量名发生冲突了。

由于在 Linux 的内核中使用了 gcc 对 C 的扩充，很自然地 Linux 的内核就只能用 gcc 编译。不仅如此，由于 gcc 和 Linux 内核在平行地发展，一旦在 Linux 内核中使用了 gcc，在其较新版本中有了新增加新扩充，就不能再使用较老版本的 gcc 来编译。也就是说，Linux 内核的各种版本有着对 gcc 版本的依赖关系。读者自然会问：“这样，Linux 内核的可移植性是否会受到损害？ ”回答是：“是的，但这是经过权衡得失利弊以后作出的决定。”首先，在可移植性与本身的质量之间，GNU 选择了以质量为优先。再说，将 gcc 移植（其实是扩充）到新的 CPU 上应非难事。回顾一下 Unix 的历史。最初的 Unix 是以汇编和 B 语言书写的，正是因为 Unix 的需要才有了 C 语言。所以，C 语言可说是 Unix 的孪生物。Unix 要发展，C 语言当然也要发展。对于 Unix 来说，C 语言不过是工具，而工具当然要服从目的本身的需要。其次，可移植性问题看似重大，其实并不太严重。如前所述，目前的 Linux 内核源代码已经支持几乎所有重要的、常用的 CPU，gcc 支持的 CPU 就更多了。而且，gcc 还支持对各种 CPU 的交叉编译。

如前所述，Linux 内核的代码中使用了大量 inline 函数。不过，这并未消除对宏操作的使用，内核中仍有许多宏操作定义。人们常常会对内核代码中一些宏操作的定义方式感到迷惑不解，有必要在这里作一些解释。先看一个实例，取自 fs/proc/kcore.c。

==================== fs/proc/kcore.c 163 163 ====================

```
163  #define DUMP_WRITE(addr,nr) do { memcpy(bufp,addr,nr); bufp += nr; } while(0)
```

读者想必知道，do-while 循环是先执行后判断循环条件。所以，这个定义意味着每当引用这个宏操作时会执行循环体一次，而且只执行一次。可是，为什么要这样通过一个 do-while 循环来定义呢？这似乎有点怪。我们不妨看看其它几种可能。首先，能不能定义成如下式样？

```
163  #define DUMP_WRITE(addr,nr) memcpy(bufp,addr,nr); bufp += nr;
```

不行。如果有一段程序在一个 if 语句中引用这个宏操作就会出问题，让我们通过一个假想的例子来说明：

```
if (addr)
    DUMP_WRITE(addr,nr);
else
    do_something_else();
```

经过预处理以后，这段代码就会变成这样：

```
if (addr)
    memcpy(bufp,addr,nr); bufp += nr;
else
    do_something_else();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

编译这段代码时 gcc 会失败，并报告语法出错。因为 gcc 认为 if 语句在 memcpy()以后就结束了，然后却又碰到一个 else。如果把 DUMP_WRITE()和 do_something_else()换一下位置，编译倒是可以通过，问题却更严重了，因为不管条件满足与否 bufp += nr 都会得到执行。

读者马上会想到要在定义中加上花括号，成为这样：

```
163 #define DUMP_WRITE(addr,nr) {memcpy(bufp,addr,nr); bufp += nr;}
```

可是，上面那段程序还是通不过编译，因为经过预处理就变成这样：

```
if (addr)
   {memcpy(bufp,addr,nr); bufp += nr;};
else
   do_something_else();
```

同样，gcc 在碰到 else 前面的“;”时就认为 if 语句已经结束，因而后面的 else 不在 if 语句中。相比之下，采用 do-while 的定义在任何情况下都没有问题。

了解了这一点之后，再来看对“空操作”的定义。由于 Linux 内核的代码要考虑到各种不同的 CPU 和不同的系统配置，所以常常需要在一定的条件下把某些宏操作定义为空操作。例如在 include/asm-i386/system.h 中的 prepare_to_switch()：

==================== include/asm-i386/system.h 14 14 ====================

```
14   #define prepare_to_switch() do { } while(0)
```

内核在调度一个进程运行，进行切换之际，在有些 CPU 上需要调用 prepare_to_switch()作些准备，而在另一些 CPU 上就不需要，所以要把它定义为空操作。

读者在学习数据结构时一定学过队列（指双链队列）操作。内核中大量地使用着队列和队列操作，而这又不是专门属于哪一个方面的内容（如进程管理、文件系统、存储管理等等），所以我们在这里作一些介绍。

如果我们有一种数据结构 foo，并且需要维持一个这种数据结构的双链队列，最简单的、也是最常用的办法就是在这个数据结构的类型定义中加入两个指针，例如：

```
typedef struct foo
{
    struct foo *prev;
    struct foo *next;
    ......
} foo;
```

然后为这种数据结构写一套用于各种队列操作的子程序。由于用来维持队列的这两个指针的类型是固定的（都指向 foo 数据结构），这些子程序不能用于其它数据结构的队列操作。换言之，需要维持多少种数据结构的队列，就得有多少套的队列操作子程序。对于使用队列较少的应用程序或许不是个大问题，但对于使用大量队列的内核就成问题了。所以，Linux 内核中采用了一套通用的、一般的、可以用到各种不同数据结构的队列操作。为此，代码的作者们把指针 prev 和 next 从具体的“宿主”数据结构中抽象出来成为一种数据结构 list_head，这种数据结构既可以“寄宿”在具体的宿主数据结构内部，成为该数据结构的一个“连接件”；也可以独立存在而成为一个队列的头。这个数据结构的定义在

PDF-XCHANGE Click to buy NOW! www.docu-track.com

include/linux/list.h 中（实际上是数据结构类型的申明，为行文方便，本书采取不那么“学究”，或者说不那么严格的态度。对“定义”和“申明”，还有对“数据结构类型”和“数据结构”，乃至“结构”这些词也常常不加严格区分。当然，我们并不鼓励读者这样做）。

==================== include/linux/list.h 16 18 ====================

```
struct list_head {
    struct list_head *next, *prev;
};
```

这里我们把结构名以粗体字排出，目的仅在于醒目，并没有特别的含义。如果需要有某种数据结构的队列，就在这种结构内部放上一个 list_head 数据结构。以用于内存页面管理的 page 数据结构为例，其定义为：（见 include/linux/mm.h）

==================== include/linux/mm.h 134 135 ====================

```
typedef struct page {
    struct list_head list;
  ......
```

==================== include/linux/mm.h 138 138 ====================

```
    struct page *next_hash;
  ......
```

==================== include/linux/mm.h 141 141 ====================

```
    struct list_head lru;
  ......
```

==================== include/linux/mm.h 148 148 ====================

```
} mem_map_t;
```

可见，在 page 数据结构中寄宿了两个 list_head 结构，或者说有两个队列操作的连接件，所以 page 结构可以同时存在于两个双链队列中。此外，结构中还有个单链指针 next_hash，用来维持一个单链的杂凑队列，不过我们在这里并不关心。

对于宿主数据结构内部的每个 list_head 数据结构都要加以初始化，可以通过一个宏操作 INIT_LIST_HEAD 进行：

==================== include/linux/list.h 25 27 ====================

```
#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)
```

参数 ptr 为指向需要初始化的 list_head 结构。可见初始化以后两个指针都指向该 list_head 结构自身。

要将一个 page 结构通过其“队列头”list 链入（有时候我们也说“挂入”）一个队列时，可以使用 list_add()，这是一个 inline 函数，其代码在 include/linux/list.h 中：

==================== include/linux/list.h 53 56 ====================

```
static __inline__ void list_add(struct list_head *new, struct list_head *head)
{
    __list_add(new, head, head->next);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

参数 new 指向欲链入队列的宿主数据结构内部的 list_head 数据结构。参数 head 则指向链入点，也是个 list_head 结构，它可以是个独守的、真正意义上的队列头，也可以在另一个宿主数据结构（甚至可以是不同类型的宿主结构）内部。这个 inline 函数调用另一个 inline 函数__list_add()来完成操作：

==================== include/linux/list.h 29 43 ====================

[list_add()>__list_add()]

```
/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static __inline__ void __list_add(struct list_head * new,
	struct list_head * prev,
	struct list_head * next)
{
	next->prev = new;
	new->next = next;
	new->prev = prev;
	prev->next = new;
}
```

对于辗转调用的函数，为帮助读者随时了解其来龙去脉，本书通常在函数的代码前面用方括号和大于号列出其调用路径，这种路径通常以一个比较重要或常用的函数为起点，例如这里就是以 list_add()为起点。不过，读者要注意，对同一函数的不同调用路径往往有很多，我们列出的只是在具体的情景或讨论中的路径。例如，有些函数也许跳过 list_add()而直接调用__list_add()，而形成另一条不同的路径。至于__list_add()本身的代码，我们就把它留给读者了。

再来看从队列中脱链的操作 list_del()：

==================== include/linux/list.h 90 93 ====================

```
static __inline__ void list_del(struct list_head *entry)
{
	__list_del(entry->prev, entry->next);
}
```

同样，这里也是调用另一个 inline 函数__list_del()来完成操作：

==================== include/linux/list.h 78 83 ====================

[list_del()>__list_del()]

```
static __inline__ void __list_del(struct list_head * prev,
				struct list_head * next)
{
	next->prev = prev;
	prev->next = next;
}
```

注意在__list_del()中的操作对象是队列中在 entry 之前和之后的两个 list_head 结构。如果 entry 是队列中的最后一项，则二者相同，就是队列的头，那也是一个 list_head 结构，不过不在任何宿主结构

PDF-XCHANGE Click to buy NOW! www.docu-track.com

内部。

读者也许已经等不及要问了：队列操作都是通过 list_head 进行的，但是那不过是个连接件，如果我们手上有个宿主结构，那当然就知道了它的某个 list_head 在哪里，从而以此为参数调用 list_add()或 list_del()；可是，反过来，当我们顺着一个队列取得其中一项的 list_head 结构时，又怎样找到其宿主结构呢？在 list_head 结构中并没有指向宿主结构的指针啊。毕竟，我们真正关心的是宿主结构，而不是连接件。

是的，这是个问题。我们还是通过一个实例来看这个问题是怎样解决的。下面是取自 mm/page_alloc.c 中的一行代码：

==================== mm/page_alloc.c 188 188 ====================

[rmqueue()]

```
188                page = memlist_entry(curr, struct page, list);
```

这里的 memlist_entry()将一个 list_head 指针 curr 换算成其宿主结构的起始地址，也就是取得指向其宿主 page 结构的指针。读者可能会对 memlist_entry()的实现和调用感到困惑。因为其调用参数 page 是个类型，而不是具体的数据。如果看一下函数 rmqueue()的整个代码，还可以发现在那里 list 竟是无定义的。

事实上，在同一文件中将 memlist_entry 定义成 list_entry，所以实际引用的是 list_entry()：

==================== mm/page_alloc.c 48 48 ====================

```
48    #define memlist_entry list_entry
```

而 list_entry 的定义则在 include/linux/list.h 中：

==================== include/linux/list.h 135 142 ====================

```
135  /**
136   * list_entry - get the struct for this entry
137   * @ptr:  the &struct list_head pointer.
138   * @type: the type of the struct this is embedded in.
139   * @member:   the name of the list_struct within the struct.
140   */
141  #define list_entry(ptr, type, member) \
142      ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

将前面的 188 行与此对照，就可以看出其中的奥秘：经过 C 预处理的文字替换，这一行的内容就成为：

```
page=((struct page*)((char*)(curr)-(unsigned long)(&((struct page*)0)->list)));
```

这里的 curr 是一个 page 结构内部的成分 list 的地址，而我们所需要的却是那个 page 结构本身的地址，所以要从地址 curr 减去一个位移量，即成分 list 在 page 内部的位移量，才能达到要求。那么，这位移量到底是多少呢？&((struct page*)0)->list 就表示当结构 page 正好在地址 0 上时其成分 list 的地址，这就是位移。

同样的道理，如果是在 page 结构的 lru 队列里，则传下来的 member 为 lru，一样能算出宿主结构的地址。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

可见，这一套操作既普遍适用，又保持了较高的效率。但是，对于阅读代码的人却有个缺点，那就是光从代码中不容易看出一个 list_head 的宿主结构是什么，而以前只要看一下指针 next 的类型就知道了。

1.5 Linux 内核源代码中的汇编语言代码

任何一个用高级语言编写的操作系统，其内核源代码中总有少部分代码是用汇编语言编写的。读过 Unix Sys V 源代码的读者都知道，在其约 3 万行的核心代码中用汇编语言编写的代码约 2000 行，分成不到 20 个扩展名.s 和.m 的文件，其中大部分是关于中断与异常处理的底层程序，还有就是与初始化有关的程序以及一些核心代码中调用的公用子程序。

用汇编语言编写核心代码中的部分代码，大体上是出于如下几个方面的考虑：

- 操作系统内核中的底层程序直接与硬件打交道，需要用到一些专用的指令，而这些指令在 C 语言中并无对应的语言成分。例如，在 386 系统结构中，对外设的输入 / 输出指令如 inb、outb 等均无对应的 C 语言语句。因此，这些底层的操作需要用汇编语言来编写。CPU 中的一些对寄存器的操作也是一样，例如，要设置一个段寄存器时，也只好用汇编语言来编写。
- CPU 中的一些特殊指令也没有对应的 C 语言成分，如关中断，开中断等等。此外，在同一种系统结构的不同 CPU 芯片中，特别是新开发出来的芯片中，往往会增加一些新的指令，例如 Pentium、Pentium II 和 Pentium MMX 都在原来的基础上扩充了新的指令，可这些指令的使用也得用汇编语言。
- 内核中实现某些操作的过程、程序段或函数，在运行时会非常频繁地被调用，因此其（时间）效率就显得很重要。而用汇编语言编写的程序，在算法和数据结构相同的条件下，其效率通常要比用高级语言编写的高。在此类程序或程序段中，往往每一条汇编指令的使用都需要经过推敲。系统调用的进入和返回就是一个典型的例子。系统调用的进出是非常频繁用到的过程，每秒钟可能会用到成千上万次，其时间效率可谓举足轻重。再说，系统调用的进出过程还牵涉到用户空间和系统空间之间的来回切换，而用于这个目的的一些指令在 C 语言中本来就没有对应的语言成分，所以，系统调用的进入和返回显然必须用汇编语言来编写。
- 在某些特殊的场合，一段程序的空间效率也会显得非常重要。操作系统的引导程序就是一个例子。系统的引导程序通常一定要能容纳在磁盘上的第一个扇区中。这时候，哪怕这段程序的大小多出一个字节也不行，所以就只能以汇编语言编写。

在 Linux 内核的源代码中，以汇编语言编写的程序或程序段，有几种不同的形式。

第一种是完全的汇编代码，这样的代码采用.s 作为文件名的后缀。事实上，尽管是“纯粹”的汇编代码，现代的汇编工具也吸收了 C 语言预处理的长处，也在汇编之前加上了一趟预处理，而预处理之前的文件则以.S 为后缀。此类（.S）文件也和 C 程序一样，可以使用#include、#ifdef 等等成分，而数据结构也一样可以在.h 文件中加以定义。

第二种是嵌入在 C 程序中的汇编语言片段。虽然在 ANSI 的 C 语言标准中并没有关于汇编片段的规定，事实上各种实际使用的 C 编译中都作了这方面的扩充，而 GNU 的 C 编译 gcc 也在这方面作了很强的扩充。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

此外，内核代码中也有几个 Intel 格式的汇编语言程序，是用于系统引导的。

由于本书专注于 Intel i386 系统结构下的 Linux 内核，下面我们只介绍 GNU 对 i386 汇编语言的支持。

对于新接触 Linux 内核源代码的读者，哪怕他比较熟悉 i386 汇编语言，在理解这两种汇编语言的程序或片段时都会感到困难，有的甚至会望而却步。其原因是：在内核“纯”汇编代码中 GNU 采用了不同于常用 386 汇编语言的句法；而在嵌入 C 程序的片段中，则更增加了一些指导汇编工具如何分配使用寄存器、以及如何与 C 程序中定义的变量相结合的语言成分。这些成分使得嵌入 C 程序中的汇编语言片段实际上变成了一种介乎 386 汇编和 C 之间的一种中间语言。

所以，我们先集中地介绍一下在内核中这两种情况下使用的 386 汇编语言，以后在具体的情景中涉及具体的汇编语言代码时还会加以解释。

1.5.1 GNU 的 386 汇编语言

在 DOS/Windows 领域中，386 汇编语言都采用由 Intel 定义的语句（指令）格式，这也是几乎在所有的有关 386 汇编语言程序设计的教科书或参考书中所使用的格式。可是，在 Unix 领域中，采用的却是由 AT&T 定义的格式。当初，当 AT&T 将 Unix 移植到 80386 处理器上时，根据 Unix 圈内人士的习惯和需要而定义了这样的格式。Unix 最初是在 PDP-11 机器上开发的，先后移植到 VAX 和 68000 系列的处理器上。这些机器的汇编语言在风格上、从而在格式上与 Intel 的有所不同。而 AT&T 定义的 386 汇编语言就比较接近那些汇编语言。后来，在 Unixware 中保留了这种格式。GNU 主要是在 Unix 领域内活动的（虽然 GNU 是“ GNU is Not Unix”的缩写）。为了与先前的各种 Unix 版本与工具有尽可能好的兼容性，由 GNU 开发的各种系统工具自然地继承了 AT&T 的 386 汇编语言格式，而不采用 Intel 的格式。

那么，这两种汇编语言之间的差距到底有多大呢？其实是大同小异。可是有时候小异也是很重要的，不加重视就会造成困扰。具体讲，主要有下面这么一些差别：

(1) 在 Intel 格式中大多使用大写字母，而在 AT&T 格式中都使用小写字母。

(2) 在 AT&T 格式中，寄存器名要加上“ %”作为前缀，而在 Intel 格式中则不带前缀。

(3) 在 AT&T 的 386 汇编语言中，指令的源操作数与目标操作数的顺序与在 Intel 的 386 汇编语言中正好相反。在 Intel 格式中是目标在前，源在后；而在 AT&T 格式中则是源在前，目标在后。例如，将寄存器 eax 的内容送入 ebx，在 Intel 格式中为“ MOVE EBX, EAX”，而在 AT&T 格式中为“ move %eax, %ebx”。看来，Intel 格式的设计者所想的是“ EBX = EAX”，而 AT&T 格式的设计者所想的是“ %eax -> %ebx”。

(4) 在 AT&T 格式中，访问指令的操作数大小（宽度）由操作码名称的最后一个字母（也就是操作码的后缀）来决定。用作操作码后缀的字母有 b（表示 8 位），w（表示 16 位）和 l（表示 32 位）。而在 Intel 格式中，则是在表示内存单元的操作数前面加上“ BYTE PTR”，“ WORD PTR”，或“ DWORD PTR”来表示。例如，将 FOO 所指内存单元中的字节取入 8 位的寄存器 AL，在两种格式中不同的表示如下：

```
MOV AL, BYTE PTR FOO    (Intel格式)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
movb FOO, %al           (AT&T 格式)
```

(5) 在 AT&T 格式中，直接操作数要加上"$"作为前缀，而在 Intel 格式中则不带前缀。所以，Intel 格式中的"PUSH 4"，在 AT&T 格式中就变为"pushl $4"。

(6) 在 AT&T 格式中，绝对转移或调用指令 jump/call 的操作数（也即转移或调用的目标地址），要加上"*"作为前缀（读者大概会联想到 C 语言中的指针吧），而在 Intel 格式中则不带。

(7) 远程的转移指令和子程序调用指令的操作码名称，在 AT&T 格式中为"ljmp"和"lcall"，而在 Intel 格式中，则为"JMP FAR"和"CALL FAR"。当转移和调用的目标为直接操作数时，两种不同的表示如下：

```
CALL FAR SECTION:OFFSET   (Intel 格式)
JMP FAR SECTION:OFFSET    (Intel 格式)

lcall $section, $offset   (AT&T 格式)
ljmp $section, $offset    (AT&T 格式)
```

与之相应的远程返回指令，则为：

```
RET FAR STACK–ADJUST      (Intel 格式)
lret $stack_adjust        (AT&T 格式)
```

(8) 间接寻址的一般格式，两者区别如下

```
SECTION:[BASE+INDEX*SCALE+DISP]     (Intel 格式)
Section:disp(base, index, scale)    (AT&T 格式)
```

这种寻址方式常常用于在数据结构数组中访问特定元素内的一个字段，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素是数据结构，则 disp 为具体字段在结构中的位移。

注意在 AT&T 格式中隐含了所进行的计算。例如，当 SECTION 省略，INDEX 和 SCALE 也省略，BASE 为 EBP，而 DISP（位移）为 4 时，表示如下：

```
[ebp-4]       (Intel 格式)
-4(%ebp)      (AT&T 格式)
```

在 AT&T 格式的括号中如果只有一项 base，就可以省略逗号，否则不能省略，所以(%ebp)相当于(%ebp, ,)，进一步相当于(%ebp, 0, 0)。又如，当 INDEX 为 EAX，SCALE 为 4（32 位），DISP 为 foo，而其它均省略，则表示为：

```
[foo+EAX*4]       (Intel 格式)
foo(, %EAX, 4)    (AT&T 格式)
```

1.5.2 嵌入 C 代码中的 386 汇编语言程序段

当需要在 C 语言的程序中嵌入一段汇编语言程序段时，可以使用 gcc 提供的"asm"语句功能。例

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

如，在 include/asm-i386/io.h 中有这么一行：

==================== include/asm-i386/io.h 40 40 ====================
```
40   #define __SLOW_DOWN_IO __asm__ __volatile__ ("outb %al, $0x80")
```

这里，暂且忽略在 asm 和 volatile 前后的两个 “ __” 字符，这也是 gcc 对 C 语言的一种扩充，后面我们还要讲到。先来看括号里面加上了引号的汇编指令。这是一条 8 位输出指令，如前所述在操作符上加了后缀“ b”以表示这是 8 位的，而 0x80 因为是常数，即所谓“直接操作数”，所以要加上前缀“ $”，而寄存器名 al 也加了前缀 “ %”。知道了前面所讲 AT&T 格式与 Intel 格式的不同，这就是一条很普通的汇编指令，很容易理解。

在同一个 asm 语句中也可以插入多行汇编程序。就在同一个文件中，在不同的条件下，__SLOW_DOWN_IO 又有不同的定义：

==================== include/asm-i386/io.h 38 38 ====================
```
38   #define __SLOW_DOWN_IO __asm__ __volatile__ ("jmp 1f \n1:\tjmp 1f \n1:)
```

这就不那么直观了。这里，一共插入了三行汇编语句，“ \n” 就是换行符，而 “ \t” 则表示 TAB 符。所以，gcc 将之翻译成下面的格式而交给 gas 去汇编：

```
        jmp 1f
     1: jmp 1f
     1:
```

这是转移指令的目标 1f 表示往前（f 表示 forward）找到第一个标号为 1 的那一行。相应地，如果是 1b 就表示往后找。这也是从早期的 Unix 汇编代码中继承下来的，读过 Unix 第 6 版的读者大概都还能记得。所以，这一小段汇编代码的用意就在于使 CPU 空做两条转移指令而消耗掉一些时间。既然是要消耗掉一些时间，而不是要节省一些时间，那么为什么要用汇编语句来实现，而不是在 C 里面来实现呢？原因在于想要对此有比较确切的控制。如果用 C 语句来消耗一些时间的话，你常常不能确切地知道经过编译以后，特别是如果经过优化的话，最后产生的汇编代码究竟怎样。

如果读者觉得这毕竟还是容易理解的话，那么下面这一段（取自 include/asm-i386/atomic.h）就困难多了：

==================== include/asm-i386/atomic.h 29 35 ====================
```
29   static __inline__ void atomic_add(int i, atomic_t *v)
30   {
31        __asm__ __volatile__(
32             LOCK "addl %1,%0"
33             :"=m" (v->counter)
34             :"ir" (i), "m" (v->counter));
35   }
```

一般而言，往 C 代码中插入汇编语言的代码片断要比“纯粹”的汇编语言代码复杂得多，因为这里有个怎样分配使用寄存器，怎样与 C 代码中的变量结合的问题。为了这个目的，必须对所用的汇编语言作更多的扩充，增加对汇编工具的指导作用。其结果是其语法实际上变成了既不同于汇编语言，也不同于 C 语言的某种中间语言。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

下面，先介绍一下插入 C 代码中的汇编成分的一般格式，并加以解释。以后，在我们走过各种情景时碰到具体的代码时还会加以提示。

插入 C 代码中的一个汇编语言代码片断可以分成四部分，以“ :”号加以分隔，其一般形式为：

```
指令部:输出部:输入部:损坏部
```

注意不要把这些“ :”号跟程序标号中所用的（如前面的 1:）混淆。

第一部分就是汇编语句本身，其格式与在汇编语言程序中使用的基本相同，但也有区别，不同之处下面会讲到。这一部分可以称为“指令部”，是必须有的，而其它各部分则可视具体的情况而省略，所以在最简单的情况下就与常规的汇编语句基本相同，如前面的两个例子那样。

当将汇编语言代码片断嵌入到 C 代码中时，操作数与 C 代码中的变量如何结合显然是个问题。在本节开头的两个例子中，汇编指令都没有产生与 C 程序中的变量结合的问题，所以比较简单。当汇编指令中的操作数需要与 C 程序中的某些变量结合时，情况就复杂多了。这是因为：程序员在编写嵌入的汇编代码时，按照程序逻辑的要求很清楚应该选用什么指令，但是却无法确切地知道 gcc 在嵌入点的前后会把哪一个寄存器分配用于哪一个变量，以及哪一个或哪几个寄存器是空闲着的。而且，光是被动地知道 gcc 对寄存器的分配情况也还是不够，还得有个手段把使用寄存器的要求告知 gcc，反过来影响它对寄存器分配。当然，如果 gcc 的功能非常强，那么通过分析嵌入的汇编代码也应该能够归纳出这些要求，再通过优化，最后也能达到目的。但是，即使那样，所引入的不确定性也还是个问题，更何况要做到这样还不容易。针对这个问题，gcc 采取了一种折中的办法：程序员只提供具体的指令，而对寄存器的使用则一般只提供一个“样板”和一些约束条件，而把到底如何与变量结合的问题留给 gcc 和 gas 去处理。

在指令部中，数字加上前缀%，如%0、%1 等等，表示需要使用寄存器的样板操作数。可以使用的此类操作数的总数取决于具体 CPU 中通用寄存器的数量。这样，指令部中用到了几个不同的这种操作数，就说明有几个变量需要与寄存器结合，由 gcc 和 gas 在编译和汇编时根据后面的约束条件自行变通处理。由于这些样板操作数也使用“ %”前缀，在涉及到具体的寄存器时就要在寄存器名前面加上两个“ %”符，以免混淆。

那么，怎样表达对变量结合的约束条件呢？这就是其余几个部分的作用。紧接在指令部后面的是“输出部”，用以规定对输出变量，即目标操作数如何结合的约束条件。每个这样的条件称为一个“约束”（constraint）。必要时输出部中可以有多个约束，互相以逗号分隔。每个输出约束以“ =”号开头，然后是一个字母表示对操作数类型的说明，然后是关于变量结合的约束。例如，在上面的例子中输出部为

```
        :"=m" (v->counter)
```

这里只有一个约束，“ =m”表示相应的目标操作数（指令部中的%0）是一个内存单元 v->counter。凡是与输出部中说明的操作数相结合的寄存器或操作数本身，在执行嵌入的汇编代码以后均不保留执行之前的内容，这就给 gcc 提供了调度使用这些寄存器的依据。

输出部后面是“输入部”。输入约束的格式与输出约束相似，但不带“ =”号。在前面例子中的输入部有两个约束。第一个为"ir" (i)，表示指令中的%1 可以是一个在寄存器中的“直接操作数”（i 表示 immediate），并且该操作数来自于 C 代码中的变量名（这里是调用参数）i。第二个约束为"m"

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

(v->countcr)，意义与输出约束中相同。如果一个输入约束要求使用寄存器，则在预处理时 gcc 会为之分配一个寄存器，并自动插入必要的指令将操作数即变量的值装入该寄存器。与输入部中说明的操作数结合的寄存器或操作数本身，在执行嵌入的汇编代码以后也不保留执行之前的内容。例如，这里的%1 要求使用寄存器，所以 gcc 会为其分配一个寄存器，并自动插入一条 movl 指令把参数 i 的数值装入该寄存器，可是这个寄存器原来的内容就不复存在了。如果这个寄存器本来就是空闲的，那倒无所谓，可是如果所有的寄存器都在使用，而只好暂时借用一个，那就得保证在使用以后恢复其原有的内容。此时 gcc 会自动在开头处插入一条 pushl 指令，将该寄存器原来的内容保存在堆栈中，而在结束以后插入一条 popl 指令，恢复寄存器的内容。

在有些操作中，除用于输入操作数和输出操作数的寄存器以外，还要将若干个寄存器用于计算或操作的中间结果。这样，这些寄存器原有的内容就损坏了，所以要在损坏部对操作的副作用加以说明，让 gcc 采取相应的措施。不过，有时候就直接把这些说明放在输出部了，那也并无不可。

操作数的编号从输出部的第一个约束（序号为 0）开始，顺序数下来，每个约束计数一次。在指令部中引用这些操作数或分配用于这些操作数的寄存器时，就在序号前面加上一个“ %”号。在指令部中引用一个操作数时总是把它当成一个 32 位的“长字”，但是对其实施的操作，则根据需要也可以是字节操作或字（16 位）操作。对操作数进行的字节操作默认为对其最低字节的操作，字操作也是一样。不过，在一些特殊的操作中，对操作数进行字节操作时也允许明确指出是对哪一个字节操作，此时在%与序号之间插入一个“ b”表示最低字节，插入一个“ h”表示次低字节。

表示约束条件的字母有很多。主要有：

“ m”，“ v” 和 “ o ” ——表示内存单元；
“ r ” ——表示任何寄存器；
“ q ” ——表示寄存器 eax，ebx，ecx，edx 之一；
“ i” 和 “ h ” ——表示直接操作数；
“ E” 和 “ F ” ——表示浮点数；
“ g ” ——表示“任意”；
“ a”，“ b”，“ c”，“ d ” ——分别表示要求使用寄存器 eax，ebx，ecx 或 edx；
“ S”，“ D ” ——分别表示要求使用寄存器 esi 或 edi；
“ I ” ——表示常数（0 至 31）。

此外，如果一个操作数要求使用与前面某个约束中所要求的是同一个寄存器，那就把与那个约束对应的操作数编号放在约束条件中。在损坏部常常会以“ memory”为约束条件，表示操作完成后内存中的内容已有改变，如果原来某个寄存器（也许在本次操作中并未用到）的内容来自内存，则现在可能已经不一致。

还要注意，当输出部为空，即没有输出约束时，如果有输入约束存在，则须保留分隔标记“ :”号。

回到上面的例子，读者现在应该可以理解这段代码的作用时将参数 i 的值加到 v->counter 上。代码中的关键字 LOCK 表示在执行 addl 指令时要把系统的总线锁住，不让别的 CPU（如果系统中有不止一个 CPU）打扰。读者也许要问，将两个数相加是很简单的操作，C 语言中明明有相应的语言成分，例如“ v->counter += i;”，为什么要用汇编呢？原因就在于，这里要求整个操作只由一条指令完成，并且要将总线锁住，以保证操作的“原子性（atomic）”。相比之下，上述的 C 语句在编译之后到底有几条指令是没有保证的，也无法要求在计算过程中对总线加锁。

再看一段嵌入汇编代码，这一次取自 include/asm-i386/bitops.h。

==================== include/asm-i386/bitops.h 18 32 ====================

```
#ifdef CONFIG_SMP
#define LOCK_PREFIX "lock ; "
#else
#define LOCK_PREFIX ""
#endif

#define ADDR (*(volatile long *) addr)

static __inline__ void set_bit(int nr, volatile void * addr)
{
        __asm__ __volatile__( LOCK_PREFIX
                "btsl %1,%0"
                :"=m" (ADDR)
                :"Ir" (nr));
}
```

这里的指令 btsl 将一个 32 位操作数中的某一位设置成 1。参数 nr 表示该位的位置。现在读者应该不感到困难，也明白为什么要用汇编语言的原因了。

再来看一个复杂一点的例子，取自 include/asm-i386/string.h：

==================== include/asm-i386/string.h 199 215 ====================

```
static inline void * __memcpy(void * to, const void * from, size_t n)
{
int d0, d1, d2;
__asm__ __volatile__(
        "rep ; movsl\n\t"
        "testb $2,%b4\n\t"
        "je 1f\n\t"
        "movsw\n"
        "1:\ttestb $1,%b4\n\t"
        "je 2f\n\t"
        "movsb\n"
        "2:"
        : "=&c" (d0), "=&D" (d1), "=&S" (d2)
        :"0" (n/4), "q" (n),"1" ((long) to),"2" ((long) from)
        : "memory");
return (to);
}
```

读者也许知道 memcpy()。这里的__memcpy()就是内核中对 memcpy()的底层实现，用来复制一块内存空间的内容，而忽略其数据结构。这是使用非常频繁的一个函数，所以其运行效率十分重要。

先看约束条件和变量与寄存器的结合。输出部有三个约束，对应于操作数%0 至%2。其中变量 d0 为操作数%0，必须放在寄存器 ecx 中，原因等一下就会明白。同样，d1 即%1 必须放在寄存器 edi 中；d2 即%2 必须放在寄存器 esi 中。再看输入部，这里有四个约束，对应于操作数%3 至%6。其中操作数

%3 与操作数%0 使用同一个寄存器，所以也必须是寄存器 ecx；并且要求由 gcc 自动插入必要的指令，事先将其设置成 n/4，实际上是将复制长度从字节个数 n 换算成长字个数 n/4。至于 n 本身，则要求 gcc 任意分配一个寄存器存放。操作数%5 与%6，即参数 to 与 from，分别与%1 和%2 使用相同的寄存器，所以也必须是寄存器 edi 和 esi。

再看指令部，读者马上就能看到这里似乎只用了%4。为什么那么多的操作数似乎都没有用呢？读完这些指令就明白了。

第一条指令是“ rep”，表示下一条指令 movsl 要重复执行，每重复一遍就把寄存器 ecx 中的内容减 1，直到变成 0 为止。所以，在这段代码中一共执行 n/4 次。那么，movsl 又干些什么呢？它从 esi 所指的地方复制一个长字到 edi 所指的地方，并使 esi 和 edi 分别加 4。这样，当代码中的 203 行执行完毕，到达 204 行时，所有的长字都已复制好，最多只剩下三个字节了。在这个过程中，实际上使用了 ecx、esi 以及 edi 三个寄存器，即%0（同时也是%3）、%2（同时也是%6）以及%1（同时也是%5）三个操作数，这些都隐含在指令中，从字面上看不出来。同时，这也说明了为什么这些操作数必须存放在指定的寄存器中。

接着就是处理剩下的三个字节了。先通过 testb 测试操作数%4，即复制长度 n 的最低字节中的 bit1，如果这一位为 1 就说明还有至少两个字节，所以通过指令 movsw 复制一个短字（esi 和 edi 则分别加 2），否则就把它跳过。再通过 testb（注意它前面是\t，表示在预处理后的汇编代码中插入一个 TAB 字符）测试操作数%4 的 bit0，如果这一位为 1 就说明还剩下一个字节，所以通过指令 movsb 再复制一个字节，否则就把它跳过。到达标号 2 的时候，执行就结束了。读者不妨自己写一段 C 代码来实现这个函数，编译以后用 objdump 看它的实现，并与此作一比较，相信就能体会到为什么这里要采用汇编语言。

作为读者的复习材料，下面是 strncmp()的代码，不熟悉 i386 指令的读者可以找一本 Intel 的指令手册对照着阅读。

==================== include/asm-i386/string.h 127 147 ====================

```
static inline int strncmp(const char * cs,const char * ct,size_t count)
{
register int __res;
int d0, d1, d2;
__asm__ __volatile__(
     "1:\tdecl %3\n\t"
     "js 2f\n\t"
     "lodsb\n\t"
     "scasb\n\t"
     "jne 3f\n\t"
     "testb %%al,%%al\n\t"
     "jne 1b\n"
     "2:\txorl %%eax,%%eax\n\t"
     "jmp 4f\n"
     "3:\tsbbl %%eax,%%eax\n\t"
     "orb $1,%%al\n"
     "4:"
              :"=a" (__res), "=&S" (d0), "=&D" (d1), "=&c" (d2)
              :"1" (cs),"2" (ct),"3" (count));
return __res;
```

```
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第2章 存储管理

2.1 Linux 内存管理的基本框架

在上一章，我们介绍了 i386 CPU，包括 Pentium，在硬件层次上对内存管理所提供的支持。内存管理最终的实现当然要由软件完成。

我们在前面谈到过，i386 CPU 中的页式存管的基本思路是：通过页面目录和页面表分两个层次实现从线性地址到物理地址的映射。这种映射模式在大多数情况下可以节省页面表所占用的空间。因为大多数进程不会用到整个虚存空间，在虚存空间中通常都留有很大的“空洞”。采用两层的方式，只要一个目录项所对应的那部分空间是个空洞，就可以把该目录项设置成“空”，从而省下了与之对应的页面表（1024 个页面描述项）。当地址的宽度为 32 位时，两层映射机制比较有效也比较合理。但是，当地址的宽度大于 32 位时，两层映射就显得不尽合理，不够有效了。

Linux 内核的设计要考虑到在各种不同 CPU 上的实现，还要考虑到在 64 位 CPU（如 Alpha）上的实现，所以不能仅仅针对 i386 结构来设计它的映射机制，而要以一种假想的、虚拟的 CPU 和 MMU（内存管理单元）为基础，设计出一种通用的模型，再把它分别落实到各种具体的 CPU 上。因此，Linux 内核的映射机制设计成三层，在页面目录和页面表中间增设了一层“中间目录”。在代码中，页面目录称为 PGD，中间目录称为 PMD，而页面表则称为 PT。PT 中的表项则称为 PTE，PTE 是“ Page Table Entry ”的缩写。PGD、PMD 和 PT 三者均为数组。相应地，在逻辑上也把线性地址从高位到低位划分成 4 个位段，各占若干位，分别用作在目录 PGD 中的下标、中间目录 PMD 中的下标、页面表中的下标以及物理页面内的位移。这样，对线性地址的映射就分成如图 2.1 所示的四步。

具体一点说，对于 CPU 发出的线性地址，虚拟的 Linux 内存管理单元分如下四步完成从线性地址到物理地址的映射：

(1) 用线性地址中最高的那一个位段作为下标在 PGD 中找到相应的表项，该表项指向相应的中间目录 PMD。
(2) 用线性地址中的第二个位段作为下标在此 PMD 中找到相应的表项，该表项指向相应页面表。
(3) 用线性地址中的第三个位段作为下标在页面表中找到相应的表项 PTE，该表项中存放的就是指向物理页面的指针。
(4) 线性地址中的最后位段为物理页面内的相对位移量，将此位移量与目标物理页面的起始地址相加便得到相应的物理地址。

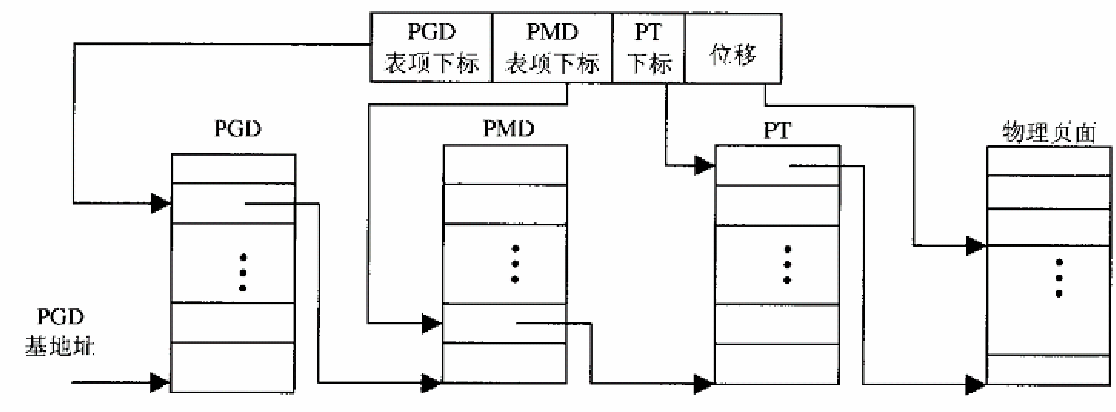


图 2.1　三层地址映射示意图

但是，这个虚拟的映射模型必须落实到具体 CPU 和 MMU 的物理映射机制。就以 i386 来说，CPU 实际上不是按三层而是按两层的模型进行地址映射的。这就需要将虚拟的三层映射落实到具体的两层映射，跳过中间的 PMD 层次。另一方面，从 Pentium Pro 开始，Intel 引入了物理地址扩充功能 PAE，允许将地址宽度从 32 位提高到 36 位，并且在硬件上支持三层映射模型。这样，在 Pentium Pro 及以后的 CPU 上，只要将 CPU 的内存管理设置成 PAE 模式，就能使虚存的映射变成三层模式。

那么，具体对于 i386 结构的 CPU，Linux 内核是怎样实现这种映射机制的呢？首先让我们来看 include/asm-i386/pgtable.h 中的一段定义：

==================== include/asm-i386/pgtable.h 106 110 ====================

```
#if CONFIG_X86_PAE
# include <asm/pgtable-3level.h>
#else
# include <asm/pgtable-2level.h>
#endif
```

根据在编译 Linux 内核之前的系统配置（config）过程中的选择，编译的时候会把目录 include/asm 符号连接到具体 CPU 专用的文件目录。对于 i386 CPU，该目录被符号连接到 include/asm-i386。同时，在配置系统时还有一个选择项是关于 PAE 的，如果所用的 CPU 是 Pentium Pro 或以上时，并且决定采用 36 位地址，则在编译时选择项 CONFIG_X86_PAE 为 1，否则为 0，根据此项选择，编译时从 pgtable-3level.h 或 pgtable-2level.h 中二者选一，前者用于 36 位地址的三层映射，而后者则用于 32 位地址的二层映射。这里，我们将集中讨论 32 位地址的二层映射。在弄清了 32 位地址的二层映射以后，读者可以自行阅读有关 36 位地址的三层映射的代码。

文件 include/asm-i386/pgtable-2level.h 中定义了二层映射中 PGD 和 PMD 的基本结构：

==================== include/asm-i386/pgtable-2level.h 4 18 ====================

```
/*
 * traditional i386 two-level paging structure:
 */

#define PGDIR_SHIFT     22
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define PTRS_PER_PGD    1024

/*
 * the i386 is two-level, so we don't really have any
 * PMD directory physically.
 */
#define PMD_SHIFT  22
#define PTRS_PER_PMD    1

#define PTRS_PER_PTE    1024
```

这里 PGDIR_SHIFT 表示线性地址中 PGD 下标位段的起始位置，文件中将其定义为 22，也即 bit22（第 23 位）。由于 PGD 是线性地址中最高的位段，所以该位段是从第 23 位到第 32 位，一共是 10 位。在文件 include/asm-i386/pgtable.h 中定义了另一个常数 PGDIR_SIZE 为：

==================== include/asm-i386/pgtable.h 117 117 ====================

```
#define PGDIR_SIZE (1UL << PGDIR_SHIFT)
```

也就是说，PGD 中的每一个表项所代表的空间（并不是 PGD 本身所占的空间）大小是 1×2^{22}。同时，pgtable_2level.h 中又定义了 PTRS_PER_PGD，也就让每个 PGD 表中指针的个数为 1024。显然，这是与线性地址中 PGD 位段的长度（10 位）相符的，因为 2^{10}=1024。这两个常数值的定义完全是针对 i386 CPU 及其 MMU 的，因为非 PAE 模式的 i386 MMU 用线性地址中的最高 10 位作为目录中的下标，而目录的大小为 1024。不过，在 32 位的系统中每个指针的大小为 4 个字节，所以 PGD 表的大小为 4KB。

对 PMD 的定义就很有意思了。PMD_SHIFT 也定义为 22，与 PGD_SHIFF 相同，表示 PMD 位段的长度为 0，一个 PMD 表项所代表的空间与 PGD 表项所代表的空间是一样大的。而 PMD 表中指针的个数 PTRS_PER_PMD 则定义为 1，表示每个 PMD 表中只有一个表项。同样，这也是针对 i386 CPU 及其 MMU 而定义的，因为要将 Linux 逻辑下的三层映射模型落实到 i386 结构物理上的二层映射，就要从线性地址逻辑上的 4 个虚拟位段中把 PMD 抽去，使它的长度为 0，所以逻辑上的 PMD 表的大小就成为 1（2^0=1）。

这样，上述的 4 步映射过程对于内核（软件）和 i386 MMU 就成为：

(1) 内核为 MMU 设置好映射目录 PGD，MMU 用线性地址中最高的那一个位段（10 位）作为下标在 PGD 中找到相应的表项。该表项逻辑上指向一个中间目录 PMD，但是是物理上直接指向相应的页面表，MMU 并不知道 PMD 的存在。

(2) PMD 只是逻辑上存在，即对内核软件在概念上存在，但是表中只有一个表项，而所谓的映射就是保持原值不变；现在一转手却指向页面表了。

(3) 内核为 MMU 设置好了所有的页面表，MMU 用线件地址中的 PT 位段作为下标在相应页面表中找到相应的表项 PTE，该表项中存放的就是指向物理页面的指针。

(4) 线性地址中的最后位段为物理页面内的相对位移量，MMU 将此位移量与目标物理页面的起始地址相加便得到相应的物理地址。

这样，逻辑上的三层映射对于 i386 CPU 和 MMU 就变成了二层映射，把中间目录 PMD 这一层跳过了，但是软件的结构却还保持着三层映射的框架。

具体的映射因空间的性质而异，但是后面读者将会看到（除用来模拟 80286 的 VM86 模式外），其

PDF-XCHANGE Click to buy NOW! www.docu-track.com

段式映射基地址总是 0，所以线性地址与虚拟地址总是一致的。在以后的讨论中，我们常常对二者不加区分。

32 位地址意味着 4G 字节的虚存空间，Linux 内核将这 4G 字节的空间分成两部分。将最高的 1G 字节（从虚地址 0xC0000000 至 0xFFFFFFFF），用于内核本身，称为“系统空间”。而将较低的 3G 字节（从虚地址 0x0 至 0xBFFFFFFF），用作各个进程的“用户空间”。这样，理论上每个进程可以使用的用户空间都是 3G 字节。当然，实际的空间大小受到物理存储器（包括内存以及磁盘交换区或交换文件）大小的限制。虽然各个进程拥有其自己的 3G 字节用户空间，系统空间却由所有的进程共享。每当一个进程通过系统调用进入了内核，该进程就在共享的系统空间中运行，不再有其自己的独立空间。从具体进程的角度看，则每个进程都拥有 4G 字节的虚存空间，较低的 3G 字节为自己的用户空间，最高的 1G 字节则为与所有进程以及内核共享的系统空间，如图 2.2 示。

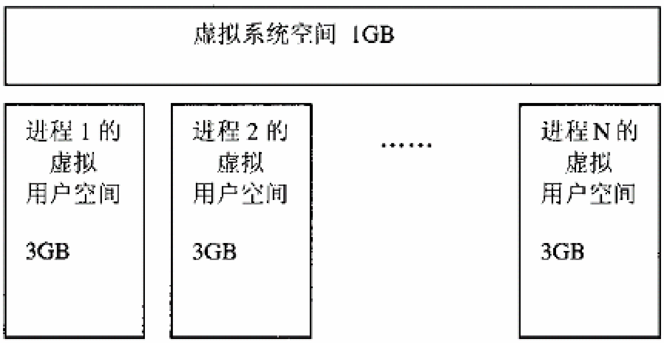


图 2.2　进程虚存空间示意图

虽然系统空间占据了每个虚存空间中最高的 1G 字节，在物理的内存中却总是从最低的地址（0）开始。所以，对于内核来说，其地址的映射是很简单的线性映射，0xC0000000 就是两者之间的位移量。因此，在代码中将此位移称为 PAGE_OFFSET 而定义于文件 include/asm-i386/page.h 中：

==================== include/asm-i386/page.h 68 82 ====================

```
/*
 * This handles the memory map.. We could make this a config
 * option, but too many people screw it up, and too few need
 * it.
 *
 * A __PAGE_OFFSET of 0xC0000000 means that the kernel has
 * a virtual address space of one gigabyte, which limits the
 * amount of physical memory you can use to about 950MB.
 *
 * If you want more physical memory than this then see the CONFIG_HIGHMEM4G
 * and CONFIG_HIGHMEM64G options in the kernel configuration.
 */

#define __PAGE_OFFSET        (0xC0000000)

......
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/asm-i386/page.h 113 116 ====================

```
113
114 #define PAGE_OFFSET        ((unsigned long)__PAGE_OFFSET)
115 #define __pa(x)            ((unsigned long)(x)-PAGE_OFFSET)
116 #define __va(x)            ((void *)((unsigned long)(x)+PAGE_OFFSET))
```

也就是说：对于系统空间而言，给定一个虚地址 x，其物理地址是从 x 中减去 PAGE_OFFSET；相应地，给定一个物理地址 x，其虚地址是 x+PAGE_OFFSET。

同时，PAGE_OFFSET 也代表着用户空间的上限，所以常数 TASK_SIZE 就是通过它定义的（include/asm-i386/processor.h）：

==================== include/asm-i386/processor.h 258 261 ====================

```
258 /*
259  * User space process size: 3GB (default).
260  */
261 #define TASK_SIZE  (PAGE_OFFSET)
```

这是因为在谈论一个用户进程的大小时，并不包括此进程在系统空间中共享的资源。

当然，CPU 并不是通过这里所说的计算方法进行地址映射的，__pa()只是为内核代码中当需要知道与一个虚地址对应的物理地址时提供方便。例如，在切换进程的时候要将寄存器 CR3 设置成指向新进程的页面目录 PGD，而该目录的起始地址在内核代码中是虚地址，但 CR3 所需要的是物理地址，这时候就要用到__pa()了。这行语句在文件 include/asm-i386/mmu_context.h 中：

==================== include/asm-i386/mmu_context.h 43 44 ====================

```
43          /* Re-load page tables */
44          asm volatile("movl %0,%%cr3": :"r" (__pa(next->pgd)));
```

这是一行汇编代码，说的是将 next->pgd，即下一个进程的页面目录起始地址，通过__pa()转换成物理地址（存放在某个寄存器），然后用 mov 指令将其写入寄存器 CR3。经过这条指令以后，CR3 就指向新进程 next 的页面目录表 PGD 了。

前面讲过，每个进程的局部段描述表 LDT 都作为一个独立的段而存在，在全局段描述表 GDT 中要有一个表项指向这个段的起始地址，并说明该段的长度以及其它一些参数。除此之外，每个进程还有一个 TSS 结构（任务状态段）也是一样。（关于 TSS 以后还会加以讨论）所以，每个进程都要在全局段描述表 GDT 中占据两个表项。那么，GDT 的容量有多大呢？段寄存器中用作 GDT 表下标的位段宽度是 13 位，所以 GDT 中可以有 8192 个描述项。除一些系统的开销（例如 GDT 中的第 2 项和第 3 项分别用于内核的代码段和数据段，第 4 项和第 5 项永远用于当前进程的代码段和数据段，第 1 项永远是 0，等等）以外，尚有 8180 个表项可供使用，所以理论上系统中最大的进程数量是 4090。

2.2 地址映射的全过程

Linux 内核采用页式存储管理。虚拟地址空间划分成固定大小的“页面”，由 MMU 在运行时将虚拟地址“映射”成（或者说变换成）某个物理内存页面中的地址。与段式存储管理相比，页式存储管

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

理有很多好处。首先，页面都是固定大小的，便于管理。更重要的是，当要将一部分物理空间的内容换出到磁盘上的时候，在段式存储管理中要将整个段（通常都很大）都换出，而在页式存储管理中则是按页进行，效率显然要高得多。页式存储管理与段式存储管理所要求的硬件支持不同，一种 CPU 既然支持页式存储管理，就无需再支持段式存储管理。但是，我们在前面讲过，i386 的情况是特殊的。由于 i386 系列的历史演变过程，它对页式存储管理的支持是在其段式存储管理已经存在了相当长的时间以后才发展起来的。所以，不管程序是怎样写的，i386 CPU 一律对程序中使用的地址先进行段式映射，然后才能进行页式映射。既然 CPU 的硬件结构是这样，Linux 内核也只好服从 Intel 的选择。这样的双重映射其实是毫无必要的，也使映射的过程变得不容易理解，以至有人还得出了 Linux 采用"段页式"存储管理技术这样一种似是而非的结论。下面读者将会看到，Linux 内核所采取的办法是使段式映射的过程实际上不起什么作用（除特殊的 VM86 模式外，那是用来模拟 80286 的）。也就是说，"你有政策，我有对策"，惹不起就躲着走。本节将通过一个情景，看看 Linux 内核在 i386 CPU 上运行时地址映射的全过程。这里要指出，这个过程仅是对 i386 处理器而言的。对于其它的处理器，比如说 M68K、Power PC 等，就根本不存在段式映射这一层了。反之，不管是什么操作系统（例如 UNIX），只要是在 i386 上实现，就必须至少在形式上要先经过段式映射，然后才可以实现其本身的设计。

假定我们写了这么一个程序：

```
#include <stdio.h>
greeting()
{
     printf("Hello, world!\n");
}

main()
{
     greeting();
}
```

读者一定很熟悉。这个程序与大部分人写的第一个 C 程序只有一点不同，我们故意让 main()调用 greeting()来显示或打印" Hello, world!"。

经过编译以后，我们得到可执行代码 hello。先来看看 gcc 和 ld（编译和连接）执行后的结果。Linux 有一个实用程序。objdump 是非常有用的，可以用来反汇编一段二进制代码。通过命令：

```
$objdump -d hello
```

可以得到我们所关心的那部分结果，输出的片断（反汇编的结果）为：

```
08048368 <greeting>:
 8048368:      55                      push   %ebp
 8048369:      89 e5                   mov    %esp,%ebp
 804836b:      83 ec 08                sub    $0x8,%esp
 804836e:      83 ec 0c                sub    $0xc,%esp
 8048371:      68 84 84 04 08          push   $0x8048484
 8048376:      e8 35 ff ff ff          call   80482b0 <printf@plt>
 804837b:      83 c4 10                add    $0x10,%esp
 804837e:      c9                      leave
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 804837f:       c3                      ret

08048380 <main>:
 8048380:       55                      push   %ebp
 8048381:       89 e5                   mov    %esp,%ebp
 8048383:       83 ec 08                sub    $0x8,%esp
 8048386:       83 e4 f0                and    $0xfffffff0,%esp
 8048389:       b8 00 00 00 00          mov    $0x0,%eax
 804838e:       83 c0 0f                add    $0xf,%eax
 8048391:       83 c0 0f                add    $0xf,%eax
 8048394:       c1 e8 04                shr    $0x4,%eax
 8048397:       c1 e0 04                shl    $0x4,%eax
 804839a:       29 c4                   sub    %eax,%esp
 804839c:       e8 c7 ff ff ff          call   8048368 <greeting>
 80483a1:       c9                      leave
 80483a2:       c3                      ret
 80483a3:       90                      nop
```

从上述结果可以看到，ld 给 greeting()分配的地址为 0x08048368。在 elf 格式的可执行代码中，ld 总是从 0x8000000 开始安排程序的“代码段”，对每个程序都是这样。至于程序在执行时在物理内存中的实际位置则就要由内核在为其建立内存映射时临时作出安排，具体地址则取决于当时所分配到的物理内存页面。

假定该程序已经在运行，整个映射机制都已经建立好，并且 CPU 正在执行 main()中的“call 08048368”这条指令，要转移到虚拟地址 0x08048368 去。接下去就请读者耐着性子跟随我们一步一步地走过这个地址的映射过程。

首先是段式映射阶段。由于地址 0x08048368 是一个程序的入口，更重要的是在执行的过程中是由 CPU 中的“指令计数器 ” EIP 所指向的，所以在代码段中。因此，i386 CPU 使用代码段寄存器 CS 的当前值来作为段式映射的“选择码”，也就是用它作为在段描述表中的下标。哪一个段描述表呢？是全局段描述表 GDT 还是局部段描述表 LDT？那就要看 CS 中的内容了。先重温一下保护模式下段寄存器的格式，见图 2.3。

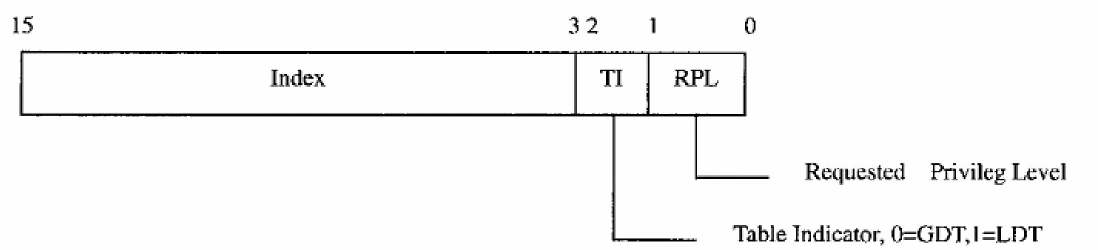


图 2.3　段寄存器格式定义

也就是说，当 bit2 为 0 时表示用 GDT，为 1 时表示用 LDT。Intel 的设计意图是内核用 GDT 而各个进程都用其自己的 LDT，最低两位 RPL 为所要求的特权级别，共分 4 级，0 为最高。

现在，可以来看看 CS 的内容了。内核在建立一个进程时都要将其段寄存器设置好（在进程管理一

PDF-XCHANGE Click to buy NOW! www.docu-track.com

章中要讲到这个问题），有关代码在 include/asm-i386/processor.h 中：

==================== include/asm-i386/processor.h 408 417 ====================

```
#define start_thread(regs, new_eip, new_esp) do {       \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));   \
        set_fs(USER_DS);                             \
        regs->xds = __USER_DS;                          \
        regs->xes = __USER_DS;                          \
        regs->xss = __USER_DS;                          \
        regs->xcs = __USER_CS;                          \
        regs->eip = new_eip;                            \
        regs->esp = new_esp;                            \
} while (0)
```

这里 regs->xds 是段寄存器 DS 的映象，余类推。这里已经可以看到一个有趣的事，就是除 CS 被设置成 USE_CS 外，其它所有的段寄存器都设置成 USER_DS。这里特别值得注意的是堆栈寄存器 SS，它也被设成 USER_DS。就是说，虽然 Intel 的意图是将一个进程的映象分成代码段、数据段和堆栈段，Linux 内核却并不买这个账。在 Linux 内核中堆栈段和数据段是不分的。

再来看看 USER_CS 和 USER_DS 到底是什么。那是在 include/asm-i386/segment.h 中定义的：

==================== include/asm-i386/segment.h 4 8 ====================

```
#define __KERNEL_CS     0x10
#define __KERNEL_DS     0x18

#define __USER_CS  0x23
#define __USER_DS  0x2B
```

也就是说，Linux 内核中只使用四种不同的段寄存器数值，两种用于内核本身，两种用于所有的进程。现在，我们将这四种数值用二进制展开并与段寄存器的格式相对照：

		Index	TI	RPL
__KERNEL_CS	0x10	0 0 0 0 0 0 0 0 0 0 0 1 0	0	0 0
__KERNEL_DS	0x18	0 0 0 0 0 0 0 0 0 0 0 1 1	0	0 0
__USER_CS	0x23	0 0 0 0 0 0 0 0 0 0 1 0 0	0	1 1
__USER_DS	0x2B	0 0 0 0 0 0 0 0 0 0 1 0 1	0	1 1

一对照就清楚了，那就是：

__KERNEL_ CS:	index = 2,	TI = 0,	RPL = 0
__KERNEL_ DS:	index = 3,	TI = 0,	RPL = 0
__USER_ CS:	index = 4,	TI = 0,	RPL = 3
__USER_ DS:	index = 5,	TI = 0,	RPL = 3

首先，TI 都是 0，也就是说全都使用 GDT。这就与 Intel 的设计意图不一致了。实际上，在 Linux

PDF-XCHANGE Click to buy NOW! www.docu-track.com

内核中基本上不使用局部段描述表 LDT。LDT 只是在 VM86 模式中运行 wine 以及其它在 Linux 上模拟运行 Windows 软件或 DOS 软件的程序中才使用。

再看 RPL，只用了 0 和 3 两级，内核为 0 级而用户（进程）为 3 级。

回到我们的程序中。我们的程序显然不属于内核，所以在进程的用户空间中运行，内核在调度该进程进入运行时，把 CS 设置成__USER_CS，即 0x23。所以，CPU 以 4 为下标，从全局段描述表 GDT 中找对应的段描述项。

初始的 GDT 内容是在 arch/i386/kernel/head.S 中定义的，其主要内容在运行中并不改变：

==================== arch/i386/kernel/head.S 444 458 ====================

```
/*
 * This contains typically 140 quadwords, depending on NR_CPUS.
 *
 * NOTE! Make sure the gdt descriptor in head.S matches this if you
 * change anything.
 */
ENTRY(gdt_table)
	.quad 0x0000000000000000	/* NULL descriptor */
	.quad 0x0000000000000000	/* not used */
	.quad 0x00cf9a000000ffff	/* 0x10 kernel 4GB code at 0x00000000 */
	.quad 0x00cf92000000ffff	/* 0x18 kernel 4GB data at 0x00000000 */
	.quad 0x00cffa000000ffff	/* 0x23 user   4GB code at 0x00000000 */
	.quad 0x00cff2000000ffff	/* 0x2b user   4GB data at 0x00000000 */
	.quad 0x0000000000000000	/* not used */
	.quad 0x0000000000000000	/* not used */
```

GDT 中的第一项（下标为 0）是不用的，这是为了防止在加电后段寄存器未经初始化就进入保护模式并使用 GDT，这也是 Intel 的规定。第二项也不用。从下标 2 至 5 共 4 项对应于前面的 4 种段寄存器数值。为便于对照，下面再次给出段描述项的格式，同时，将 4 个段描述项的内容按二进制展开如下：

```
K_CS:  0000 0000 1100 1111 1001 1010 0000 0000  0000 0000 0000 0000 1111 1111 1111 1111
K_DS:  0000 0000 1100 1111 1001 0010 0000 0000  0000 0000 0000 0000 1111 1111 1111 1111
K_CS:  0000 0000 1100 1111 1111 1010 0000 0000  0000 0000 0000 0000 1111 1111 1111 1111
K_DS:  0000 0000 1100 1111 1111 0010 0000 0000  0000 0000 0000 0000 1111 1111 1111 1111
```

读者结合下页图 2.4 段描述项的定义仔细对照，可以得出如下结论：

(1) 四个段描述项的下列内容都是相同的。

- B0-B15、B16-B31 都是 0 ——基地址全为 0；
- L0-L15、L16-L19 都是 1 ——段的上限全是 0xfffff；
- C 位都是 1 ——段长单位均为 4KB；
- D 位都是 1 ——对四个段的访问都是 32 位指令
- P 位都是 1 ——四个段都在内存。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

结论：每个段都是从 0 地址开始的整个 4GB 虚存空间，虚地址到线性地址的映射保持原值不变。因此，讨论或理解 Linux 内核的页式映射时，可以直接将线性地址当作虚拟地址，二者完全一致。

(2) 有区别的地方只是在 bit40～bit46，对应于描述项中的 type 以及 S 标志和 DPL 位段。

- 对 KERNEL_CS：DPL=0，表示 0 级；S 位为 1，表示代码段或数据段：type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 KERNEL_DS：DPL=0，表示 0 级；S 位为 1，表示代码段或数据段；type 为 0010 表示数据段，可读，可写，尚未受到访问。
- 对 USER_CS：DPL=3，表示 3 级：S 位为 1，表示代码段或数据段；type 为 1010，表示代码段，可读，可执行，尚未受到访问。
- 对 USER_DS：即下标为 5 时，DPL=3，表示 3 级：S 位为 1，表示代码段或数据段，type 为 0010，表示数据段，可读，可写，尚未受到访问。

有区别的其实只有两个地方：一是 DPL，内核为最高的 0 级，用户为最低的 3 级；另一个是段的类型，或为代码，或为数据。这两项都是 CPU 在映射过程中要加以检查核对的。如果 DPL 为 0 级，而段寄存器 CS 中的 DPL 为 3 级，那就不允许了，因为那说明 CPU 的当前运行级别比想要访问的区段要低。或者，如果段描述项说是数据段，而程序中通过 CS 来访问，那也不允许。实际上，这里所作的检查比对在页式映射的过程中还要进行，所以既然用了页式映射，这里的检查比对就是多余。要不是 i386 CPU 中的 MMU 要作这样的检查比对，那就只要一个段描述项就够了。进一步，要不是 i386 CPU 中的 MMU 规定先作段式映射，然后才可以作页式映射，那就根本不需要段描述项和段寄存器了。所以，这里 Linux 内核只不过是装模作样地糊弄 i386 CPU，对付其检查比对而已。

读者也许会问：如此说来，怀有恶意的程序员岂不是可以通过设置寄存器 CS 和 DS，甚至连这也不用，就可以打破 i386 的段式保护机制吗？是的，但是不要忘记，Linux 内核之所以这样安排，原因在于它采用的是页式存储管理，这里只不过是在对付本来就毫无必要却又非得如此的例行公事而已。真正重要的是页式映射阶段的保护机制。

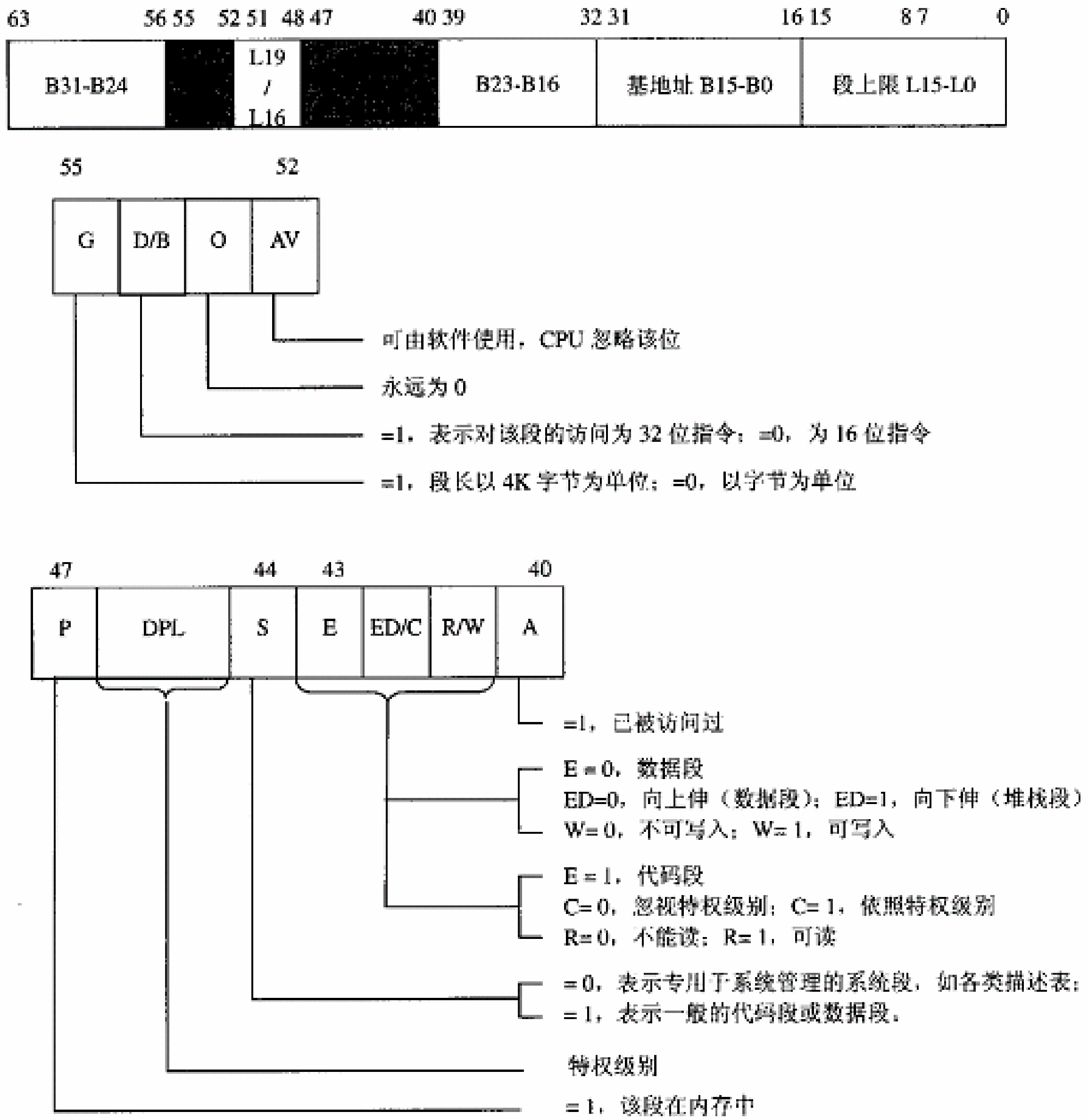


图 2.4 段描述项定义

所以，Linux 内核设计的段式映射机制把地址 0x08048368 映射到了其自身，现在作为线性地址出现了。下面才进入了页式映射的过程。

与段式映射过程中所有进程全都共用一个 GDT 不一样，现在可是动真格的了，每个进程都有其自身的页面目录 PGD，指向这个目录的指针保持在每个进程的 mm_struct 数据结构中。每当调度一个进程进入运行的时候，内核都要为即将运行的进程设置好控制寄存器 CR3，而 MMU 的硬件则总是从 CR3 中取得指向当前页面目录的指针。不过，CPU 在执行程序时使用的是虚拟地址，而 MMU 硬件在进行映射时所用的则是物理地址。这是在 inline 函数 switch_mm() 中完成的，其代码见

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

include/asm-i386/mmu_context.h。但是我们在此关心的只是其中最关键的一行：

==================== include/asm-i386/mmu_context.h 28 29 ====================

```
static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk, unsigned cpu)
{
  ......
```

==================== include/asm-i386/mmu_context.h 44 44 ====================

```
        asm volatile("movl %0,%%cr3": :"r" (__pa(next->pgd)));
  ......
```

==================== include/asm-i386/mmu_context.h 59 59 ====================

```
}
```

我们以前曾用这行代码说明__pa()的用途，这里将下一个进程的页面目录 PGD 的物理地址装入寄存器%%cr3，也即 CR3。细心的读者可能会问：这样，在这一行以前和以后 CR3 的值不一样，也就是使用不同的页面目录，不会使程序的执行不能连续了吗？答案是，这是在内核中。不管什么进程，一旦进入内核就进了系统空间，都有相同的页面映射，所以不会有问题。

当我们在程序中要转移到地址 0X08048368 去的时候，进程正在运行中，CR3 早已设置好，指向我们这个进程的页面目录了。先将线性地址 0X08048368 按二进制展开：

```
0000  1000  0000  0100  1000  0011  0110  1000
```

对照线性地址的格式，可见最高 10 位为二进制的 0000100000，也就是十进制的 32，所以 i386 CPU（确切地说是 CPU 中的 MMU，下同）就以 32 为下标去页面目录中找到其目录项。这个目录项中的高 20 位指向一个页面表。CPU 在这 20 位后边添上 12 个 0 就得到该页面表的指针。以前我们讲过，每个页面表占一个页面，所以自然就是 4K 字节边界对齐的，其起始地址的低 12 位一定是 0。正因为如此，才可以把 32 位目录项中的低 12 位挪作它用，其中的最低位为 P 标志位，为 1 时表示该页面表在内存中。

找到页面表以后，CPU 再来看线性地址中的中间 10 位。线性地址 0X08048368 的第二个 10 位为 0001001000，即十进制的 72。于是 CPU 就以此为下标在已经找到的页面表中找到相应的表项。与目录项相似，当页面表项的 P 标志位为 1 时表示所映射的页面在内存中。32 位的页面表项中的高 20 位指向一个物理内存页面，在后边添上 12 个 0 就得到这物理内存页面的起始地址。所不同的是，这一次指向的不再是一个中间结构，而是映射的目标页面了。在其起始地址上加上线性地址中的最低 12 位，就得到了最终的物理内存地址。这时这个线性地址的最低 12 位为 0x368。所以，如果目标页面的起始地址为 0x740000 的话（具体取决于内核中的动态分配），那么 greeting()入口的物理地址就是 0x740368，greeting()的执行代码就存储在这里。

读者可能已经注意到，在页面映射的过程中，i386 CPU 要访问内存三次。第一次是页面目录，第二次是页面表，第三次才是访问真正的目标。所以虚存的高效实现有赖于高速缓存（cache）的实现。有了高速缓存，虽然在第一次用到具体的页面目录和页面表时要到内存中去读取，但一旦装入了高速缓存以后，一般都可以在高速缓存中找到，而不需要再到内存中去读取了。另一方面，这整个过程是由硬件实现的，所以速度很快。

除常规的页式映射之外，为了能在 Linux 内核上仿真运行采用段式存储管理的 Windows 或 DOS 软

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

件，还提供了两个特殊的、与段式存储管理有关的系统调用。

2.2.1 modify_ldt(int func, void *ptr, unsigned bytecount)

这个系统调用可以用来改变当前进程的局部段描述表。在自由软件基金会 FSF 下面，除 Linux 以外还有许多个项目在进行。其中有一个叫“ WINE”，其名字来自“ Windows Emulation”，目的是在 Linux 上仿真运行 Windows 的软件。多年来，有些 Windows 软件已经广泛地为人们所接受和熟悉（如 MS Word 等），而在 Linux 上没有相同的软件往往成了许多人不愿意转向 Linux 的原因。所以，在 Linux 上建立一个环境，使得用户可以在上面运行 Windows 的软件，就成了一个开拓市场的举措。而系统调用 modify_ldt()就是因开发 WINE 的需要而设置的。当 func 参数的值为 0 时，该调用返回本进程局部段描述表的实际大小，而表的内容就在用户通过 ptr 提供的缓冲区中。当 func 参数的值为 1 时，ptr 应指向一个结构 modify_ldt_ldt_s。而 bytecount 则为 sizeof(struct modify_ldt_ldt_s)。该数据结构的定义见于 include/asm-i386/ldt.h：

==================== include/asm-i386/ldt.h 15 25 ====================

```
struct modify_ldt_ldt_s {
      unsigned int  entry_number;
      unsigned long base_addr;
      unsigned int  limit;
      unsigned int  seg_32bit:1;
      unsigned int  contents:2;
      unsigned int  read_exec_only:1;
      unsigned int  limit_in_pages:1;
      unsigned int  seg_not_present:1;
      unsigned int  useable:1;
};
```

其中 entry_number 是想要改变的表项的序号，即下标。而结构中其余的成分则给出要设置到各个位段中去的值。

读者可能会要问：这样岂不是在内存管理机制上挖了个洞？既然一个进程可以改变它的局部段描述表，它岂不就可设法侵犯到其它进程或内核的空间中去？这要从两方面来看。一方面它确实是在内存管理机制上开了一个小小的缺口，但另一方面它的背后仍然是 Linux 内核的页式存储管理，只要不让用户进程掌握修改页面目录和页面表的手段，系统就还是安全的。

2.2.2 vm86(struct vm86_struct *info)

与 modify_ldt()相类似，还有一个系统调用 vm86()，用来在 linux 上模拟运行 DOS 软件。i386 CPU 专门提供了一种寻址方式 VM86，用来在保护模式下模拟运行实模式（real-mode）的软件。其目的是为采用保护模式的系统（如 Windows，0S/2 等）提供与实模式软件（常常是 DOS 软件）的兼容性。事至如今，需要加以模拟运行 DOS 软件已经很少了，或者干脆已经绝迹了。所以本书在 80386 的寻址方

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

式一节中略去了 VM86 模式的内容，有兴趣的读者可以参照 Intel 的技术资料，自行阅读内核中有关的源代码，主要有 include/asm-i386/vm86.h 和 arch/i386/kemel/vm86.c。

显然，这两个系统调用以及由此实现的功能实际上并不属于 Linux 内核本身的存储管理框架，而是为了与 Windows 软件和 DOS 软件兼容而采取的权宜之计。

2.3 几个重要的数据结构和函数

从硬件的角度来说，Linux 内核只要能为硬件准备好页面目录 PGD、页面表 PT 以及全局段描述表 GDT 和局部段描述表 LDT，并正确地设置有关的寄存器，就完成了内存管理机制中地址映射部分的准备工作。虽然最终的目的是地址映射，但是实际上内核所需要做的管理工作却要复杂得多。在与内存管理有关的内核代码中，有几个数据结构是很重要的，这些数据结构及其使用构成了代码中内存管理的基本框架。

页面目录 PGD、中间目录 PMD 和页面表 PT 分别是由表项 pgd_t、pmd_t 以及 pte_t 构成的数组，而这些表项又都是数据结构，定义于 include/asm-i386/page.h 中：

==================== include/asm-i386/page.h 36 50 ====================

```
/*
 * These are used to make use of C type-checking..
 */
#if CONFIG_X86_PAE
typedef struct { unsigned long pte_low, pte_high; } pte_t;
typedef struct { unsigned long long pmd; } pmd_t;
typedef struct { unsigned long long pgd; } pgd_t;
#define pte_val(x) ((x).pte_low | ((unsigned long long)(x).pte_high << 32))
#else
typedef struct { unsigned long pte_low; } pte_t;
typedef struct { unsigned long pmd; } pmd_t;
typedef struct { unsigned long pgd; } pgd_t;
#define pte_val(x) ((x).pte_low)
#endif
#define PTE_MASK   PAGE_MASK
```

可见，当采用 32 位地址时，pgd_t、pmd_t 和 pte_t 实际上就是长整数，而当采用 36 位地址时则是 long long 整数。之所以不直接定义成长整数的原因在于这样可以让 gcc 在编译时加以更严格的类型检查。同时，代码中又定义了几个简单的函数来访问这些结构的成分；如 pte_val()、pgd_val()等（难怪有人说 Linux 内核的代码吸收了面向对象的程序设计手法）。但是，如我们以前说过的那样，表项 PTE 作为指针实际上只需要它的高 20 位。同时，所有的物理页面都是跟 4K 字节的边界对齐的，因而物理页面起始地址的高 20 位又可以看作是物理页面的序号。所以，pte_t 中的低 12 位用于页面的状态信息和访问权限。在内核代码中并没有在 pte_t 等结构中定义有关的位段，而是在 include/asm-i386/page.h 中另行定义了一个用来说明页面保护的结构 pgprot_t：

==================== include/asm-i386/page.h 52 52 ====================

```
typedef struct { unsigned long pgprot; } pgprot_t;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

参数 pgprot 的值与 i386 MMU 的页面表项的低 12 位相对应，其中 9 位是标志位，表示所映射页面的当前状态和访问权限（详见第 1 章）。内核代码中作了相应的定义：

==================== include/asm-i386/pgtable.h 162 172 ====================

```
#define _PAGE_PRESENT  0x001
#define _PAGE_RW   0x002
#define _PAGE_USER 0x004
#define _PAGE_PWT  0x008
#define _PAGE_PCD  0x010
#define _PAGE_ACCESSED 0x020
#define _PAGE_DIRTY    0x040
#define _PAGE_PSE  0x080    /* 4 MB (or 2MB) page, Pentium+, if present.. */
#define _PAGE_GLOBAL   0x100    /* Global TLB entry PPro+ */

#define _PAGE_PROTNONE 0x080    /* If not present */
```

注意这里的_PAGE_PROTNONE 对应于页面表项中的 bit7，在 Intel 的手册中说这一位保留不用，所以对 MMU 不起作用。

在实际使用中，pgprot 的数值总是小于 0x1000，而 pte 中的指针部分则总是大于 0x1000，将二者合在一起就得到实际用于页面表中的表项。具体的计算是由 include/asm-i386/pgtable-2level.h 中定义的宏操作 mk_pte 完成的：

==================== include/asm-i386/pgtable-2level.h 61 61 ====================

```
#define __mk_pte(page_nr,pgprot) __pte(((page_nr) << PAGE_SHIFT) | pgprot_val(pgprot))
```

这里将页面序号左移 12 位，再与页面的控制 / 状况位段相或，就得到了表项的值。这里引用的两个宏操作均定义于 include/asm-i386/page.h 中：

==================== include/asm-i386/page.h 56 58 ====================

```
#define pgprot_val(x)  ((x).pgprot)
#define __pte(x) ((pte_t) { (x) } )
```

内核中有个全局量 mem_map，是一个指针，指向一个 page 数据结构的数组（下面会讨论 page 结构），每个 page 数据结构代表着一个物理页面，整个数组就代表着系统中的全部物理页面。因此，页面表项的高 20 位对于软件和 MMU 硬件有着不同的意义。对于软件，这是一个物理页面的序号，将这个序号用作下标就可以从 mem_map 找到代表这个物理页面的 page 数据结构。对于硬件，则（在低位补上 12 个 0 后）就是物理页面的起始地址。

还有一个常用的宏操作 set_pte()，用来把一个表项的值设置到一个页面表项中，这个宏操作定义于 include/asm-i386/pgtable-2level.h 中：

==================== include/asm-i386/pgtable-2level.h 42 42 ====================

```
#define set_pte(pteptr, pteval) (*(pteptr) = pteval)
```

在映射的过程中，MMU 首先检查的是 P 标志位，就是上面的_PAGE_PRESENT，它指示着所映射的页面是否在内存中。只有在 P 标志位为 1 的时候 MMU 才会完成映射的全过程；否则就会因不能完成映射而产生一次缺页异常，此时表项中的其它内容对 MMU 就没有任何意义了。除 MMU 硬件根据

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

页面表项的内容进行页面映射外，软件也可以设置或检测页面表项的内容，上面的 set_pte()就是用来设置页面表项。内核中还为检测页面表项的内容定义了一些工具性的函数或宏操作，其中最重要的有：

==================== include/asm-i386/pgtable-2level.h 60 60 ====================
```
#define pte_none(x)         (!(x).pte_low)
```

==================== include/asm-i386/pgtable.h 248 248 ====================
```
#define pte_present(x) ((x).pte_low & (_PAGE_PRESENT | _PAGE_PROTNONE))
```

对软件来说，页面表项为 0 表示尚未为这个表项（所代表的虚存页面）建立映射，所以还是空白；而如果页面表项不为 0，但 P 标志位为 0，则表示映射已经建立，但是所映射的物理页面不在内存中（已经换出到交换设备上，详见后面的页面交换）。

==================== include/asm-i386/pgtable.h 269 271 ====================
```
static inline int pte_dirty(pte_t pte)        { return (pte).pte_low & _PAGE_DIRTY; }
static inline int pte_young(pte_t pte)        { return (pte).pte_low & _PAGE_ACCESSED; }
static inline int pte_write(pte_t pte)        { return (pte).pte_low & _PAGE_RW; }
```

当然，这些标志位只有在 P 标志位为 1 时才有意义。

如前所述，当页面表项的 P 标志位为 1 时，其高 20 位为相应物理页面起始地址的高 20 位，由于物理页面的起始地址必然是与页面边界对齐的，所以低 12 位一定是 0。如果把整个物理内存看成一个物理页面的“数组”，那么这高 20 位（右移 12 位以后）就是数组的下标，也就是物理页面的序号。相应地，用这个下标，就可以在上述的 page 结构数组中找到代表目标物理页面的数据结构。代码中为此也定义了一个宏操作（include/asm-i386/pgtable-2level.h）：

==================== include/asm-i386/pgtable-2level.h 59 59 ====================
```
#define pte_page(x)         (mem_map+((unsigned long)(((x).pte_low >> PAGE_SHIFT))))
```

由于 mem_map 是 page 结构指针，操作的结果也是个 page 结构指针，mem_map+x 与&mem_map[x]是一样的。在内核的代码中，还常常需要根据虚存地址找到相应物理页面的 page 数据结构，所以还为此也定义了一个宏操作（include/asm-i386/page.h）：

==================== include/asm-i386/page.h 117 117 ====================
```
#define virt_to_page(kaddr) (mem_map + (__pa(kaddr) >> PAGE_SHIFT))
```

代表物理页面的 page 数据结构是在文件 include/linux/mm.h 中定义的：

==================== include/linux/mm.h 126 148 ====================
```
/*
 * Try to keep the most commonly accessed fields in single cache lines
 * here (16 bytes or greater).  This ordering should be particularly
 * beneficial on 32-bit processors.
 *
 * The first line is data used in page cache lookup, the second line
 * is used for linear searches (eg. clock algorithm scans).
 */
typedef struct page {
     struct list_head list;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        struct address_space *mapping;
        unsigned long index;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;    /* atomic flags, some possibly updated asynchronously */
        struct list_head lru;
        unsigned long age;
        wait_queue_head_t wait;
        struct page **pprev_hash;
        struct buffer_head * buffers;
        void *virtual; /* non-NULL if kmapped */
        struct zone_struct *zone;
} mem_map_t;
```

内核中用来表示这个数据结构的变量名常常是 page 或 map。

当页面的内容来自一个文件时，index 代表着该页面在文件中的序号；当页面的内容被换出到交换设备上、但还保留着内容作为缓冲时，则 index 指明了页面的去向。结构中各个成分的次序是有讲究的，目的是尽量使得联系紧密的若干成分在执行时被装填入高速缓存的同一缓冲线（16 个字节）中。

系统中的每一个物理页面都有一个 page 结构（或 mem_map_t）。系统在初始化时根据物理内存的大小建立起一个 page 结构数组 mem_map，作为物理页面的“仓库”，里面的每个 page 数据结构都代表着系统中的一个物理页面。每个物理页面的 page 结构在这个数组里的下标就是该物理页面的序号。“仓库”里的物理页面划分成 ZONE_DMA 和 ZONE_NORMAL 两个管理区（根据系统配置，还可能有第三个管理区 ZONE_HIGHMEM，用于物理地址超过 1GB 的存储空间）。

管理区 ZONE_DMA 里的页面是专供 DMA 使用的。为什么供 DMA 使用的页面要单独加以管理呢？首先，DMA 使用的页面是磁盘 I/O 所必需的，如果把仓库中所有的物理页面都分配光了，那就无法进行页面与盘区的交换了。此外，还有些特殊的原因。在 i386 CPU 中，页式存储管理的硬件支持是在 CPU 内部实现的，而不像另有些 CPU 那样由一个单独的 MMU 提供，所以 DMA 不经过 MMU 提供的地址映射。这样，外部设备就要直接提供访问物理页面的地址，可是有些外设（特别是插在 ISA 总线上的外设接口卡）在这方面往往有些限制，要求用于 DMA 的物理地址不能过高。另一方面，正因为 DMA 不经过 MMU 提供的地址映射，当 DMA 所需的缓冲区超过一个物理页面的大小时，就要求两个页面在物理上连续，因为此时 DMA 控制器不能依靠在 CPU 内部的 MMU 将连续的虚存页面映射到物理上不连续的页面。所以，用于 DMA 的物理页面是要单独加以管理的。

每个管理区都有一个数据结构，即 zone_struct 数结构。在 zone_struct 数据结构中有一组“空闲区间”（free_area_t）队列。为什么是“一组”队列，而不是“一个”队列呢？这也是因为常常需要成“块”地分配在物理空间内连续的多个页面，所以要按块的大小分别加以管理。因此，在管理区数据结构中既要有一个队列来保持一些离散（连续长度为 1）的物理页面，还要有一个队列来保持一个连续长度为 2 的页面块以及连续长度为 4、8、16、…、直至 2^{MAX_ORDER} 的页面块。常数 MAX_ORDER 定义为 10，也就是说最大的连续页面块可以达到 2^{10}=1024 个页面，即 4M 字节。这两个数据结构以及几个常数都是在文件 include/linux/mmzone.h 中定义的：

==================== include/linux/mmzone.h 11 58 ====================

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Free memory management - zoned buddy allocator.
 */

#define MAX_ORDER 10

typedef struct free_area_struct {
     struct list_head   free_list;
     unsigned int       *map;
} free_area_t;

struct pglist_data;

typedef struct zone_struct {
     /*
      * Commonly accessed fields:
      */
     spinlock_t         lock;
     unsigned long      offset;
     unsigned long      free_pages;
     unsigned long      inactive_clean_pages;
     unsigned long      inactive_dirty_pages;
     unsigned long      pages_min, pages_low, pages_high;

     /*
      * free areas of different sizes
      */
     struct list_head   inactive_clean_list;
     free_area_t        free_area[MAX_ORDER];

     /*
      * rarely used fields:
      */
     char           *name;
     unsigned long      size;
     /*
      * Discontig memory support fields.
      */
     struct pglist_data *zone_pgdat;
     unsigned long      zone_start_paddr;
     unsigned long      zone_start_mapnr;
     struct page        *zone_mem_map;
} zone_t;

#define ZONE_DMA        0
#define ZONE_NORMAL         1
#define ZONE_HIGHMEM        2
#define MAX_NR_ZONES        3
```

管理区结构中的 offset 表示该分区在 mem_map 中的起始页面号。一旦建立起管理区，每个物理页

PDF-XCHANGE Click to buy NOW! www.docu-track.com

面便永久地属于某一个管理区，具体取决于页面的起始地址，就好像一幢建筑物属于哪一个派出所管辖取决于其地址一样。空闲区 free_area_struct 结构中用来维持双向链队列的结构 list_head 是一个通用的数据结构，linux 内核中需要使用双向链队列的地方都使用这种数据结构。结构很简单，就是 prev 和 next 两个指针。回到上面的 page 结构，其中的第一个成分就是一个 list_head 结构，物理页面的 page 结构正是通过它进入 free_area_struct 结构中的双向链队列的。在“物理页面的分配”一节中，我们将讲述内核怎样从它的仓库中分配一块物理空间，即若干连续的物理页面。

在传统的计算机结构中，整个物理空间都是均匀一致的，CPU 访问这个空间中的任何一个地址所需的时间都相同，所以称为“均质存储结构”（Uniform Memory Architecture），简称 UMA。可是，在一些新的系统结构中，特别是在多 CPU 结构的系统中，物理存储空间在这方面的一致性却成了问题。试想有这么一种系统结构：

- 系统的中心是一条总线，例如 PCI 总线。
- 有多个 CPU 模块连接在系统总线上，每个 CPU 模块都有本地的物理内存，但是也可以通过系统总线访问其它 CPU 模块上的内存。
- 系统总线上还连接着一个公用的存储模块，所有的 CPU 模块都可以通过系统总线来访问它。
- 所有这些物理内存的地址互相连续而形成一个连续的物理地址空间。

显然，就某个特定的 CPU 而言，访问其本地的存储器是速度最快的，而穿过系统总线访问公用存储模块或其它 CPU 模块上的存储器就比较慢，而且还面临因可能的竞争而引起的不确定性。也就是说，在这样的系统中，其物理存储空间虽然地址连续，“质地”却不一致，所以称为“非均质存储结构”（Non-Uniform Memory Architecture），简称 NUMA。在 NUMA 结构的系统中，分配连续的若干物理页面时一般都要求分配在质地相同的区间（称为 node，即“节点”）。举例来说，要是 CPU 模块 1 要求分配 4 个物理页面，可是由于本模块上的空间已经不够，所以前 3 个页面分配在本模块上，而最后一个页面却分配到了 CPU 模块 2 上，那显然是不合适的。在这样的情况下，将 4 个页面都分配在公用模块上显然要好得多。

事实上，严格意义上的 UMA 结构几乎是不存在的。就拿配置最简单的单 CPU 的 PC 来说，其物理存储空间就包括了 RAM、ROM（用于 BIOS），还有图形卡上的静态 RAM。但是在 UMA 结构中，除“主存”RAM 以外的存储器都很小，所以把它们放在特殊的地址上成为小小的“孤岛”，再在编程时特别加以注意就可以了。然而，在典型的 NUMA 结构中就需要来自内核中内存管理机制的支持了。由于多处理器结构的系统日益广泛的应用，Linux 内核 2.4.0 版提供了对 NUMA 的支持（作为一个编译可选项）。

由于 NUMA 结构的引入，对于上述的物理页面管理机制也作了相应的修正。管理区不再是属于最高层的机构，而是在每个存储节点中都有至少两个管理区。而且前述的 page 结构数组也不再是全局性的，而是从属于具体的节点了。从而，在 zone_struct 结构（以及 page 结构数组）之上又有了另一层代表着存储节点的 pglist_data 数据结构，定义于 include/linux/mmzone.h 中：

==================== include/linux/mmzone.h 79 90 ====================

```
typedef struct pglist_data {
      zone_t node_zones[MAX_NR_ZONES];
      zonelist_t node_zonelists[NR_GFPINDEX];
      struct page *node_mem_map;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        unsigned long *valid_addr_bitmap;
        struct bootmem_data *bdata;
        unsigned long node_start_paddr;
        unsigned long node_start_mapnr;
        unsigned long node_size;
        int node_id;
        struct pglist_data *node_next;
    } pg_data_t;
```

显然，若干存储节点的 pglist_data 数据结构可以通过指针 node_next 形成一个单链队列。每个结构中的指针 node_mem_map 指向具体节点的 page 结构数组，而数组 node_zones[]就是该节点的最多三个页面管理区。反过来，在 zone_struct 结构中也有一个指针 zone_pgdat，指向所属节点的 pglist_data 数据结构。

同时，又在 pglist_data 结构里设置了一个数组 node_zonelists[]，其类型定义也在同一文件中：

==================== include/linux/mmzone.h 71 74 ====================

```
typedef struct zonelist_struct {
        zone_t * zones [MAX_NR_ZONES+1]; // NULL delimited
        int gfp_mask;
} zonelist_t;
```

这里的 zones[]是个指针数组，各个元素按特定的次序指向具体的页面管理区，表示分配页面时先试 zones[0]所指向的管理区，如不能满足要求就试 zones[1]所指向的管理区，等等。这些管理区可以属于不同的存储节点。这样，针对上面所举的例子就可以规定：先试本节点，即 CPU 模块 1 的 ZONE_DMA 管理区，若不够 4 个页面就全部从公用模块的 ZONE_DMA 管理区中分配。就是说，每个 zonelist_t 规定了一种分配策略。然而，每个存储节点不应该只有一种分配策略，所以在 pglist_data 结构中提供的是一个 zonelist_t 数组，数组的大小为 NR_GFPINDEX，定义为：

==================== include/linux/mmzone.h 76 76 ====================

```
#define NR_GFPINDEX         0x100
```

就是说，最多可以规定 256 种不同的策略。要求分配页面时，要说明采用哪一种分配策略。

前面几个数据结构都是用于物理空间管理的，现在来看看虚拟空间的管理，也就是虚存页面的管理。虚存空间的管理不像物理空间的管理那样有一个总的物理页面仓库，而是以进程为基础的，每个进程都有各自的虚存（用户）空间。不过，如前所述，每个进程的“系统空间”是统一为所有进程所共享的。以后我们对进程的“虚存空间”和“用户空间”这两个词常常会不加区分。

如果说物理空间是从“供”的角度来管理的，也就是：“仓库中还有些什么”；则虚存空间的管理是从“需”的角度来管理的，就是“我们需要用虚存空间中的哪些部分”。拿虚存空间中的“用户空间”部分来说，大概没有一个进程会真的需要使用全部的 3G 字节的空间。同时，一个进程所需要使用的虚存空间中的各个部位又未必是连续的，通常形成若干离散的虚存“区间”。很自然地，对虚存空间的抽象是一个重要的数据结构。在 Linux 内核中，这就是 vm_area_struct 数据结构，定义于 include/linux/mm.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/mm.h 35 69 ====================

```
/*
 * This struct defines a memory VMM memory area. There is one of these
 * per VM-area/task.  A VM area is any part of the process virtual memory
 * space that has a special rule for the page-fault handlers (ie a shared
 * library, the executable area etc).
 */
struct vm_area_struct {
     struct mm_struct * vm_mm;   /* VM area parameters */
     unsigned long vm_start;
     unsigned long vm_end;

     /* linked list of VM areas per task, sorted by address */
     struct vm_area_struct *vm_next;

     pgprot_t vm_page_prot;
     unsigned long vm_flags;

     /* AVL tree of VM areas per task, sorted by address */
     short vm_avl_height;
     struct vm_area_struct * vm_avl_left;
     struct vm_area_struct * vm_avl_right;

     /* For areas with an address space and backing store,
      * one of the address_space->i_mmap{,shared} lists,
      * for shm areas, the list of attaches, otherwise unused.
      */
     struct vm_area_struct *vm_next_share;
     struct vm_area_struct **vm_pprev_share;

     struct vm_operations_struct * vm_ops;
     unsigned long vm_pgoff;     /* offset in PAGE_SIZE units, *not* PAGE_CACHE_SIZE */
     struct file * vm_file;
     unsigned long vm_raend;
     void * vm_private_data;     /* was vm_pte (shared mem) */
};
```

在内核的代码中，用于这个数据结构的变量名常常是 vma。

结构中的 vm_start 和 vm_end 决定了一个虚存区间。vm_start 是包含在区间内的，而 vm_end 则不包含在区间内。区间的划分并不仅仅取决于地址的连续性，也取决于区间的其它属性，主要是对虚存页面的访问权限。如果一个地址范围内的前一半页面和后一半页面有不同的访问权限或其它属性，就得要分成两个区间。所以，包含在同一个区间里的所有页面都应有相同的访问权限（或者说保护属性）和其它一些属性，这就是结构中的成分 vm_page_prot 和 vm_flags 的用途。属于同一个进程的所有区间都要按虚存地址的高低次序链接在一起，结构中的 vm_next 指针就是用于这个目的。由于区间的划分并不仅仅取决了地址的过续性，一个进程的虚存（用户）空间很可能会被划分成大量的区间。内核中给定一个虚拟地址而要找出其所属的区间是一个频繁用到的操作，如果每次都要顺着 vm_next 在链中

PDF-XCHANGE Click to buy NOW! www.docu-track.com

作线性搜索的话，势必会显著地影响到内核的效率。所以，除了通过 vm_next 指针把所有区间串成一个线性队列以外，还可以在区间数量较大时为之建立一个 AVL（Adelson-Velskii and Landis）树。AVL 树是一种平衡的树结构，读者从有关的数据结构专著中可以了解到，在 AVL 树中搜索的速度快而代价是 O(lg n)，即与树的大小的对数（而不是树的大小）成比例。虚存区间结构 vm_area_struct 中的 vm_avl_height、vm_avl_left 以及 vm_avl_right 三个成分就是用于 AVL 树，表示本区间在 AVL 树中的位置的。

在两种情况下虚存页面（或区间）会跟磁盘文件发生关系。一种是盘区交换（swap），当内存页面不够分配时，一些久未使用的页面可以被交换到磁盘上去，腾出物理页面以供更急需的进程使用，这就是大家所知道的一般意义上的“按需调度”页式虚存管理（demand paging）。另一种情况则是将一个磁盘文件映射到一个进程的用户空间中。Linux 提供了一个系统调用 mmap()（实际上是从 Unix Sys V R4.2 开始的），使一个进程可以将一个已经打开的文件映射到其用户空间中，此后就可以像访问内存中一个字符数组那样来访问这个文件的内容，而不必通过 lseek()、read()或 write()等进行文件操作。

由于虚存区间（最终是页面）与磁盘文件的这种联系，在 vm_area_struct 结构中相应地设置了一些成分，如 mapping、vm_next_share、vm_pprev_share、vm_file 等，用以记录和管理此种联系。我们将在以后结合具体的情景介绍这些成分的使用。

虚存区间结构中另一个重要的成分是 vm_ops，这是指向一个 vm_operations_struct 数据结构的指针。这种数据结构也是在 include/linux/mm.h 中定义的：

==================== include/linux/mm.h 115 124 ====================

```
/*
 * These are the virtual MM functions - opening of an area, closing and
 * unmapping it (needed to keep files on disk up-to-date etc), pointer
 * to the functions called when a no-page or a wp-page exception occurs.
 */
struct vm_operations_struct {
     void (*open)(struct vm_area_struct * area);
     void (*close)(struct vm_area_struct * area);
     struct page * (*nopage)(struct vm_area_struct * area, unsigned long address, int write_access);
};
```

结构中全是函数指针。其中 open、close、nopage 分别用于虚存区间的打开、关闭和建立映射。为什么要有这些函数呢？这是因为对于不同的虚存区间可能会需要一些不同的附加操作。函数指针 nopage 指示当因（虚存）页面不在内存中而引起“页面出错”（page fault）异常（见第 3 章）时所应调用的函数。

最后，vm_area_struct 中还有一个指针 vm_mm，该指针指向一个 mm_struct 数据结构，那是在 include/linux/sched.h 中定义的：

==================== include/linux/sched.h 203 227 ====================

```
struct mm_struct {
     struct vm_area_struct * mmap;         /* list of VMAs */
     struct vm_area_struct * mmap_avl;     /* tree of VMAs */
     struct vm_area_struct * mmap_cache;   /* last find_vma result */
     pgd_t * pgd;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        atomic_t mm_users;          /* How many users with user space? */
        atomic_t mm_count;          /* How many references to "struct mm_struct" (users count as 1) */
        int map_count;                  /* number of VMAs */
        struct semaphore mmap_sem;
        spinlock_t page_table_lock;

        struct list_head mmlist;        /* List of all active mm's */

        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        unsigned long arg_start, arg_end, env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        unsigned long cpu_vm_mask;
        unsigned long swap_cnt;/* number of pages to swap on next pass */
        unsigned long swap_address;

        /* Architecture-specific MM context */
        mm_context_t context;
    };
```

在内核的代码中，用于这个数据结构（指针）的变量名常常是 mm。

显然，这是在比 vm_area_struct 更高层次上使用的数据结构。事实上，每个进程只有一个 mm_struct 结构，在每个进程的“进程控制块”，即 task_struct 结构中，有一个指针指向该进程的 mm_struct 结构。可以说，mm_struct 数据结构是进程整个用户空间的抽象，也是总的控制结构。结构中的头三个指针都是关于虚存区间的。第一个 mmap 用来建立一个虚存区间结构的单链线性队列。第二个 mmap_avl 用来建立一个虚存区间结构的 AVL 树，这在前面已经谈过。第三个指针 mmap_cache，用来指向最近一次用到的那个虚存区间结构；这是因为程序中用到的地址常常带有局部性，最近一次用到的区间很可能就是下一次要用到的区间，这样就可以提高效率。另一个成分 map_count，则说明在队列中（或 AVL 树中）有几个虚存区间结构，也就是说该进程有几个虚存区间。指针 pgd 显而易见是指向该进程的页面目录的，当内核调度一个进程进入运行时，就将这个指针转换成物理地址，并写入控制寄存器 CR3，这在前面已经看到过了。另一方面，由于 mm_struct 结构及其下属的 vm_area_struct 结构都有可能在不同的上下文中受到访问，而这些访问又必须互斥，所以在结构中设置了用于 P、V 操作的信号量（semaphore），即 mmap_sem。此外，page_table_lock 也是为类似的目的而设置的。

虽然一个进程只使用一个 mm_struct 结构，反过来一个 mm_struct 结构却可能为多个进程所共享。最简单的例子就是，当一个进程创建（vfork()或 clone()，见第 4 章）一个子进程时，其子进程就可能与父进程共享一个 mm_struct 结构。所以，在 mm_struct 结构中还为此设置了计数器 mm_users 和 mm_count。类型 atomic_t 实际上就是整数，但是对这种类型的整数进行的操作必须是“原子”的，也就是不允许因中断或其它原因而受到干扰。

指针 segment 指向该进程的局部段描述表 LDT。个过，一般的进程是不用局部段描述表的，只有在 VM86 模式下才会有 LDT。

结构中其它成分的用途比较显而易见，如 start_code、end_code、start_data、end_data 等等就是该进程映象中代码段、数据段、存储堆以及堆栈段的起点和终点，这里就不多说了。注意，不要把进程

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

映象中的这些“段”跟“段式存储管理”中的“段”相混淆。

如前所述，mm_struct 结构及其属下的各个 vm_area_struct 只是表明了对虚存空间的需求。一个虚拟地址有相应的虚存区间存在，并不保证该地址所在的页面已经映射到某一个物理（内存或盘区）页面，更不保证该页面就在内存中。当一个未经映射的页面受到访问时，就会产生一个“ Page Fault”异常（也称缺页异常、缺页中断），那时候 Page Fault 异常的服务程序就会来处理这个问题。所以，从这个意义上，mm_struct 和 vm_area_struct 说明了对页面的需求；前面的 page、zone_struct 等结构则说明了对页面的供应；而页面目录、中间目录以及页面表则是二者中间的桥梁。

图 2.5 是个示意图，图中说明了用于进程虚存管理的各种数据结构之间的联系。

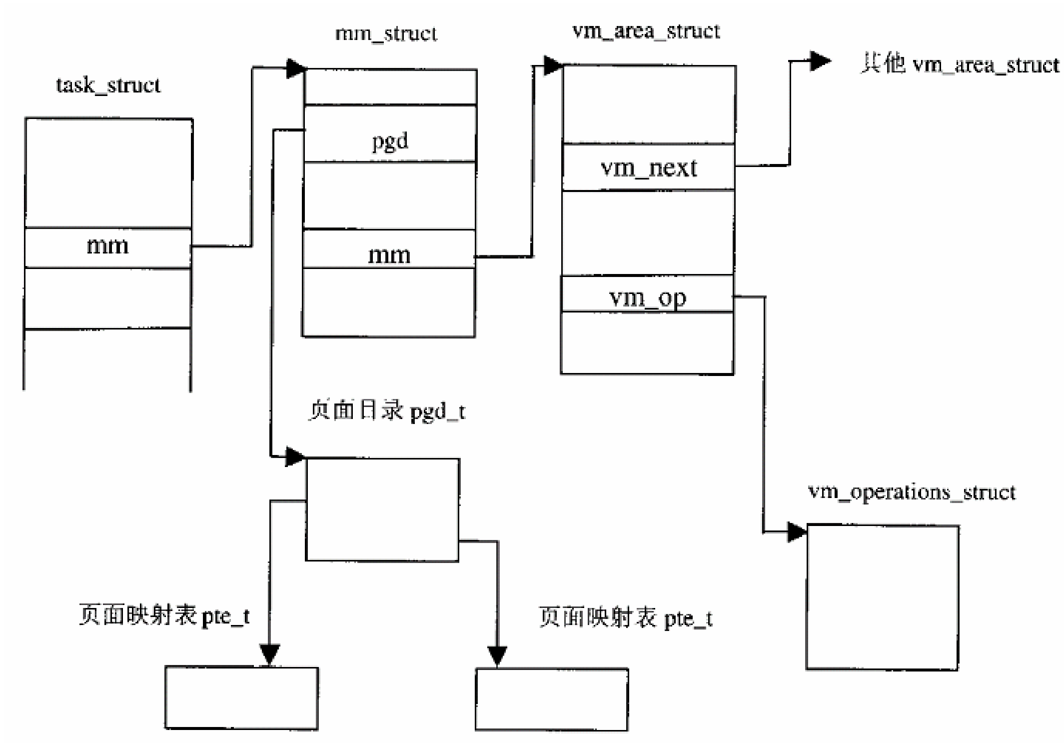


图 2.5　虚存管理数据结构联系图

前面讲过，在内核中经常要用到这样的操作：给定一个属于某个进程的虚拟地址，要求找到其所属的区间以及相应的 vm_area_struct 结构。这是由 find_vma()来实现的，其代码在 mm/mmap.c 中：

==================== mm/mmap.c 404 440 ====================

```
/* Look up the first VMA which satisfies  addr < vm_end,  NULL if none. */
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
    struct vm_area_struct *vma = NULL;

    if (mm) {
        /* Check the cache first. */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /* (Cache hit rate is typically around 35%.) */
        vma = mm->mmap_cache;
        if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
            if (!mm->mmap_avl) {
                /* Go through the linear list. */
                vma = mm->mmap;
                while (vma && vma->vm_end <= addr)
                    vma = vma->vm_next;
            } else {
                /* Then go through the AVL tree quickly. */
                struct vm_area_struct * tree = mm->mmap_avl;
                vma = NULL;
                for (;;) {
                    if (tree == vm_avl_empty)
                        break;
                    if (tree->vm_end > addr) {
                        vma = tree;
                        if (tree->vm_start <= addr)
                            break;
                        tree = tree->vm_avl_left;
                    } else
                        tree = tree->vm_avl_right;
                }
            }
            if (vma)
                mm->mmap_cache = vma;
        }
    }
    return vma;
}
```

当我们说到一个特定的用户空间虚拟地址时，必须说明是哪一个进程的虚存空间中的地址，所以函数的参数有两个，一个是地址，一个是指向该进程的 mm_struct 结构的指针。首先看一下这地址是否恰好在上一次（最近一次）访问过的同一个区间中。根据代码作者所加的注释，命中率一般可达 35%，这也正是在 mm_struct 结构中设置一个 mmap_cache 指针的原因。如果没有命中的话，那就要搜索了。如果已经建立过 AVL 结构（指针 mmap_avl 非零），就在 AVL 树中搜索，否则就在线性队列中搜索。最后，如果找到的话，就把 mmap_cache 指针设置成指向所找到的 vm_area_struct 结构。函数的返回值为零（NULL），表示该地址所属的区间还未建立。此时通常就得要建立起一个新的虚存区间结构，再调用 insert_vm_struct()将其插入到 mm_struct 中的线性队列或 AVL 树中去。函数 insert_vm_struct()的源代码在同一文件中：

==================== mm/mmap.c 961 968 ====================

```
void insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *vmp)
{
    lock_vma_mappings(vmp);
    spin_lock(&current->mm->page_table_lock);
    __insert_vm_struct(mm, vmp);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    spin_unlock(&current->mm->page_table_lock);
    unlock_vma_mappings(vmp);
}
```

将一个 vm_area_struct 数据结构插入队列的操作实际是由__insert_vm_struct()完成的，但是这个操作绝不允许受到干扰，所以要对操作加锁。这里加了两把锁。第一把加在代表新区间的 vm_area_struct 数据结构中，第二把加在代表着整个虚存空间的 mm_struct 数据结构中，使得在操作过程中不让其它进程能够在中途插进来，也对这两个数据结构进行队列操作。下面是__insert_vm_struct()的主体，我们略去了与文件映射有关的部分代码。由于与 find_vma()很相似，这里就不加说明了，留给读者自行阅读。对 AVL 缺乏了解的读者可以只阅读不采用 AVL 树，即 mm->mmap_avl 为 0 的那一部分代码。

==================== mm/mmap.c 913 939 ====================

```
/* Insert vm structure into process list sorted by address
 * and into the inode's i_mmap ring.  If vm_file is non-NULL
 * then the i_shared_lock must be held here.
 */
void __insert_vm_struct(struct mm_struct *mm, struct vm_area_struct *vmp)
{
    struct vm_area_struct **pprev;
    struct file * file;

    if (!mm->mmap_avl) {
         pprev = &mm->mmap;
         while (*pprev && (*pprev)->vm_start <= vmp->vm_start)
              pprev = &(*pprev)->vm_next;
    } else {
         struct vm_area_struct *prev, *next;
         avl_insert_neighbours(vmp, &mm->mmap_avl, &prev, &next);
         pprev = (prev ? &prev->vm_next : &mm->mmap);
         if (*pprev != next)
              printk("insert_vm_struct: tree inconsistent with list\n");
    }
    vmp->vm_next = *pprev;
    *pprev = vmp;

    mm->map_count++;
    if (mm->map_count >= AVL_MIN_MAP_COUNT && !mm->mmap_avl)
         build_mmap_avl(mm);

  ......
```

==================== mm/mmap.c 959 959 ====================

```
}
```

当一个虚存空间中区间的数量较小时，在线性队列中搜索的效率并不成为问题，所以不需要为之建立 AVL 树。而当区间的数量增大到 AVL_MIN_MAP_COUNT，即 32 时，就需要通过 build_mmap_avl()建立 AVL 树，以提高搜索效率了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

2.4 越界访问

页式存储管理机制通过页面目录和页面表将每个线性地址（也可以理解为虚拟地址）转换成物理地址。如果在这个过程中遇到某种阻碍而使 CPU 无法最终访问到相应的物理内存单元，映射便失败了，而当前的指令也就不能执行完成。此时 CPU 会产生一次页面出错（Page Fault）异常（Exception）（也称缺页中断），进而执行预定的页面异常处理程序，使应用程序得以从因映射失败而暂停的指令处开始恢复执行，或进行一些善后处理。这里所说的阻碍可以有以下几种情况：

- 相应的页面目录项或页面表项为空，也就是该线性地址与物理地址的映射关系尚未建立，或者已经撤销。
- 相应的物理页面不在内存中。
- 指令中规定的访问方式与页面的权限不符，例如企图写一个“只读”的页面。

在这个情景里，我们假定一段用户程序曾经将一个已打开文件通过 mmap()系统调用映射到内存，然后又已经将映射撤销（通过 munmap()系统调用）。在撤销一个映射区间时，常常会在虚存地址空间中留下一个孤立的空洞，而相应的地址则不应继续使用了。但是，在用户程序中往往会有错误，以致在程序中某个地方还再次访问这个已经撤销的区域（程序员们一定会同意，这是不足为奇的）。这时候，一次因越界访问一个无效地址（Invalid Address）而引起映射失败，从而就产生了一次页面出错异常。中断请求以及异常的响应机制将在“中断和异常”一章中集中介绍，读者在那里可以找到从发生异常到进入内核相应服务程序的全过程。这里假定 CPU 的运行已经到达了页面异常服务程序的主体 do_page_fault()的入口处。

函数 do_page_fault()的代码在文件 arch/i386/mm/fault.c 中。这个函数的代码比较长，我们将随着情景的进展按需要来展示其有关的片断。这里先来看开头几行代码：

==================== arch/i386/mm/fault.c 106 152 ====================

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
	struct task_struct *tsk;
	struct mm_struct *mm;
	struct vm_area_struct * vma;
	unsigned long address;
	unsigned long page;
	unsigned long fixup;
	int write;
	siginfo_t info;

	/* get the address */
	__asm__("movl %%cr2,%0":"=r" (address));

	tsk = current;

	/*
	 * We fault-in kernel-space virtual memory on-demand. The
	 * 'reference' page table is init_mm.pgd.
```

```
	 *
	 * NOTE! We MUST NOT take any locks for this case. We may
	 * be in an interrupt or a critical region, and should
	 * only copy the information from the master page table,
	 * nothing more.
	 */
	if (address >= TASK_SIZE)
		goto vmalloc_fault;

	mm = tsk->mm;
	info.si_code = SEGV_MAPERR;

	/*
	 * If we're in an interrupt or have no user
	 * context, we must not take the fault..
	 */
	if (in_interrupt() || !mm)
		goto no_context;

	down(&mm->mmap_sem);

	vma = find_vma(mm, address);
	if (!vma)
		goto bad_area;
	if (vma->vm_start <= address)
		goto good_area;
	if (!(vma->vm_flags & VM_GROWSDOWN))
		goto bad_area;
```

首先是一行汇编码。为什么要用汇编码呢？当 i386 CPU 产生“页面错”异常时，CPU 将导致映射失败的线性地址放在控制寄存器 CR2 中，而这显然是相应的服务程序所必需的信息。可是，在 C 语言中并没有相应的语言成分可以用来读取 CR2 的内容，所以只能用汇编代码。这行汇编代码只有输出部而没有输入部，它将%0 与变量 address 相结合，并说明该变量应该被分配在一个寄存器中。

同时，内核的中断 / 异常响应机制还传过来两个参数。一个是 pt_regs 结构指针 regs，它指向例外发生前夕 CPU 中各寄存器内容的一份副本，这是由内核的中断响应机制保存下来的“现场”。而 error_code 则进一步指明映射失败的具体原因。

然后是获取当前进程的 task_struct 数据结构。在内核中，可以通过一个宏操作 current 取得当前进程（当前正在运行的进程）的 task_struct 结构的地址。在每个进程的 task_struct 结构中有一个指针，指向其 mm_struct 数据结构，而跟虚存管理和映射有关的信息都在那个结构中。这里要指出，CPU 实际进行的映射并不涉及 mm_struct 结构，而是像以前讲过的那样通过页面目录和页面表进行，但是 mm_struct 结构反映了，或者说描述了这种映射。

接下来，需要检测两个特殊情况。一个特殊情况是 in_interrupt()返回非 0，说明映射的失败发生在某个中断服务程序中，因而与当前进程毫无关系。另一个特殊情况是当前进程的 mm 指针为空，也就是说该进程的映射尚未建立，当然也就不可能与当前进程有关。可是，不跟当前进程有关，in_interrupt()又返回 0，那这次异常发生在什么地方呢？其实还是在某个中断 / 异常服务程序中，只不过不在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

in_interrupt()能检测到的范围中而已。如果发生这些特殊情况，控制就通过 goto 语句转到标号 no_cotext 处，不过那与我们这个情景无关，所以我们略去对那段代码的讨论。

以下的操作有互斥的要求，也就是不容许别的进程来打扰，所以要有对信号量的 P/V 操作，即 down()/up()操作来保证。为了这个目的，在 mm_struct 结构中还设置了所需的信号量 mmap_sem。这样，从 down()返回以后，就不会有别的进程来打扰了。

可以想像，在知道了发生映射失败的地址以及所属的进程以后，接下来应该要搞清楚的是这个地址是否落在某个已经建立起映射的区间，或者进一步具体指出在哪个区间。事实正是这样，这就是 find_vma()所要做的事情。以前讲过，find_vma()试图在一个虚存空间中找出结束地址大于给定地址的第一个区间。如果找不到的话，那本次页面异常就必定是因越界访问而引起。那么，在什么情况下会找不到呢？回忆一下内核对用户虚存空间的使用，堆栈在用户区的顶部，从上向下伸展，而进程的代码和数据都是自底向上分配空间。如果没有一个区间的结束地址高于给定的地址，那就是说明这个地址是在堆栈之下，也就是 3G 字节以上了。要从用户空间访问属于系统的空间，那当然是越界了，然后就转向 bad_area，不过我们这个情景所说的不是这个情况。

如果找到了这么一个区间，而且其起始地址又不高于给定的地址（见程序 148 行），那就说明给定的地址恰好落在这个区间。这样，映射肯定已经建立，所以就转向 good_area 去进一步检查失败的原因。这也不是我们这个情景所要说的。

除了这两种情况，剩下的就是给定地址正落在两个区间当中的空洞里，也就是该地址所在页面的映射尚未建立或已经撤销。在用户虚存空间中，可能有两种不同的空洞。第一种空洞只能有一个，那就是在堆栈区以下的那个大空洞，它代表着供动态分配（通过系统调用 brk()）而仍未分配出去的空间。当映射失败的地址落在这个空洞里时，还有个特殊情况要考虑，我们将在下一个情景中讨论。但是，怎样才知道这地址是落在这个空洞里呢？请看程序 150 行。我们知道，堆栈区是向下伸展的，如果 find_vma()找到的区间是堆栈区间，那么在它的 vm_flags 中应该有个标志位 VM_GROWSDOWN。要是该标志位为 0 的话，那就说明空洞上方的区间并非堆栈区，说明这个空洞是因为一个映射区间被撤销而留下的，或者在建立映射时跳过了一块地址。这就是第二种可能，也是我们这个情景所说的情况。

所以，我们就随着这里的 goto 语句转向 bad_area，那是在 224 行：

==================== arch/i386/mm/fault.c 220 239 ====================

[do_page_fault()]

```
220  /*
221   * Something tried to access memory that isn't in our memory map..
222   * Fix it, but check if it's kernel or user first..
223   */
224  bad_area:
225       up(&mm->mmap_sem);
226
227  bad_area_nosemaphore:
228       /* User mode accesses just cause a SIGSEGV */
229       if (error_code & 4) {
230            tsk->thread.cr2 = address;
231            tsk->thread.error_code = error_code;
232            tsk->thread.trap_no = 14;
233            info.si_signo = SIGSEGV;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
234             info.si_errno = 0;
235             /* info.si_code has been set above */
236             info.si_addr = (void *)address;
237             force_sig_info(SIGSEGV, &info, tsk);
238             return;
239         }
```

首先，当控制流到达这里时，已经不再需要互斥（因为不再对 mm_struct 结构进行操作），所以通过 up()退出临界区。接着，就要进一步考察 error_code，看看失败的具体原因。代码的作者为此加了注解：

==================== arch/i386/mm/fault.c 96 105 ====================

```
96   /*
97    * This routine handles page faults.  It determines the address,
98    * and the problem, and then passes it off to one of the appropriate
99    * routines.
100   *
101   * error_code:
102   *    bit 0 == 0 means no page found, 1 means protection fault
103   *    bit 1 == 0 means read, 1 means write
104   *    bit 2 == 0 means kernel, 1 means user-mode
105   */
```

当 error_code 的 bit2 为 1 时，表示失败是当 CPU 处于用户模式时发生的，这正与我们的情景相符，所以控制将进入 229 行。在那里，对当前进程的 task_struct 结构内的一些成分进行一些设置以后，就向该进程发出一个强制的“信号”（或称“软中断”）SIGSEGV。至此，本次例外服务就结束了。

读者大概会问：“就这样完了？”是的，完了。接下来的详情，读者在看了有关中断处理和信号的章节以后就会明白。每次从中断 / 异常返回之前，都要检查当前进程是否有悬而未决的信号需要处理，在我们这个情景里当然是有的，其中至少有一个就是 SIGSEGV。然后，内核根据这些待处理信号的性质以及进程本身的选择决定怎么办。对有些软中断的处理是“自愿”的，有些则是强制的。而对于 SIGSEGV 的反应，那是强制的，其后果是在该进程的显示屏上显示程序员们怕见到却又经常见到的“Segment Fault”提示，然后使进程流产（撤销）。至于从异常处理返回用户空间后的地址，在这种情况下并无意义，因为本来就不会回去了。

我们在这里跳过了 do_page_fault()中的许多代码，因为那些代码与我们眼下这个特定的情景无关。不过，以后在其它的情景里我们还会回到这些代码中来。

2.5 用户堆栈的扩展

在上一个情景中，我们“游览参观”了一次因越界访问而造成映射失败从而引起进程流产的过程。但是，读者也许会感到惊奇，越界访问有时候是正常的。不过，这只发生在一种消况下。现在我们就来看看当用户堆栈过小，但是因越界访问而“因祸得福”得以伸展的情景。在阅读本情景之前，读者应该先温习一下前一个情景。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

假设在进程运行的过程中，已经用尽了为本进程分配的堆栈区间，也就是从堆栈的“顶部”开始（记住，堆栈是从上向下伸展的），已经到达了已映射的堆栈区间的下沿。或者说，CPU 中的堆栈指针%esp 已经指向堆栈区间的起始地址，见图 2.6。

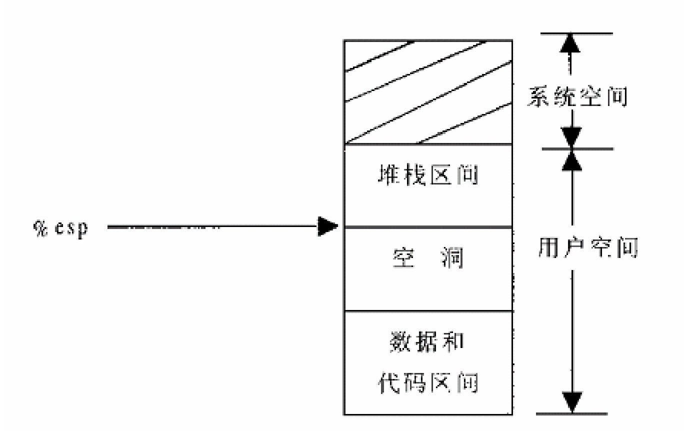


图 2.6　进程地址空间示意图

假定现在需要调用某个子程序，因此 CPU 需将返回地址压入堆栈，也就是要将返回地址写入虚存空间中地址为(%esp - 4)的地方。可是，在我们这个情景中地址(%esp - 4)落入了空洞中，这是尚未映射的地址，因此必然要引起一次页面错异常。让我们顺着上一个情景中已经走过的路线到达文件 arch/i386/mm/fault.c 的第 151 行。

==================== arch/i386/mm/fault.c 151 164 ====================

[do_page_fault()]

```
151      if (!(vma->vm_flags & VM_GROWSDOWN))
152          goto bad_area;
153      if (error_code & 4) {
154          /*
155           * accessing the stack below %esp is always a bug.
156           * The "+ 32" is there due to some instructions (like
157           * pusha) doing post-decrement on the stack and that
158           * doesn't show up until later..
159           */
160          if (address + 32 < regs->esp)
161              goto bad_area;
162      }
163      if (expand_stack(vma, address))
164          goto bad_area;
```

这一次，空洞上方的区间是堆栈区间，其 VM_GROWSDOWN 标志位为 1，所以 CPU 就继续往前执行。当映射失败发生在用户空间（bit2 为 1）时，因堆栈操作而引起的越界是作为特殊情况对待的，所以还需要检查发生异常时的地址是否紧挨着堆栈指针所指的地方。在我们这个情景中，那是%esp-4，当然是紧挨着的。但是如果是%esp-40 呢？那就不会是因为正常的堆栈操作而引起，而是货真价实的非法越界访问了。可是，怎样来判定“正常”或不正常呢？通常，一次压入堆栈的是 4 个字节，所以该

地址应该是%esp-4。但是 i386 CPU 有一条 pusha 指令，可以一次将 32 个字节（8 个 32 位寄存器的内容）压入堆栈。所以，检查的准则是%esp-32。超出这个范围就一定是错的了，所以跟在前一个情景中一样，转向 bad_area。而在我们现在这个情景中，这个测试应是顺利通过了。

既然是属于正常的堆栈扩展要求，那就应该从空洞的顶部开始分配若干页面建立映射，并将之并入难栈区间，使其得以扩展。所以就要调用 expand_stack()，这是在文件 include/linux/mm.h 中定义的一个 inline 函数：

==================== include/linux/mm.h 487 504 ====================
[do_page_fault()>expand_stack()]
```
/* vma is the first one with  address < vma->vm_end,
 * and even  address < vma->vm_start. Have to extend vma. */
static inline int expand_stack(struct vm_area_struct * vma, unsigned long address)
{
	unsigned long grow;

	address &= PAGE_MASK;
	grow = (vma->vm_start - address) >> PAGE_SHIFT;
	if (vma->vm_end - address > current->rlim[RLIMIT_STACK].rlim_cur ||
	    ((vma->vm_mm->total_vm + grow) << PAGE_SHIFT) > current->rlim[RLIMIT_AS].rlim_cur)
		return -ENOMEM;
	vma->vm_start = address;
	vma->vm_pgoff -= grow;
	vma->vm_mm->total_vm += grow;
	if (vma->vm_flags & VM_LOCKED)
		vma->vm_mm->locked_vm += grow;
	return 0;
}
```

参数 vma 指向一个 vm_area_struct 数据结构，代表着一个区间，在这里就是代表着用户空间堆栈所在的区间。首先，将地址按页面边界对齐，并计算需要增长几个页面才能把给定的地址包括进去（通常是一个）。这里还有个问题，堆栈的这种扩展是否不受限制，直到把空间中的整个空洞用完为止呢？不是的。每个进程的 task_struct 结构中都有个 rlim 结构数组，规定了对每种资源分配使用的限制，而 RLIMIT_STACK 就是对用户空间堆栈大小的限制。所以，这里就进行这样的检查。如果扩展以后的区间大小超过了可用于堆栈的资源，或者使动态分配的页面总量超过了可用于该进程的资源限制，那就不能扩展了，就会返回一个负的出错代码-ENOMEM，表示没有存储空间可以分配了；否则就应返回 0。当 expand_stack()返回的值为非 0，也即-ENOMEM 时，在 do_page_fault()中也会转向 bad_area，其结果就与前一情景一样了。不过一般情况下都不至于用尽资源，所以 expand_stack()一般都是正常返回的。但是，我们已经看到，expand_stack()只是改变了堆栈区的 vm_area_struct 结构，而并未建立起新扩展的页面对物理内存的映射。这个任务由接下去的 good_area 完成：

==================== arch/i386/mm/fault.c 165 207 ====================
[do_page_fault()]
```
/*
 * Ok, we have a good vm_area for this memory access, so
 * we can handle it..
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
168  */
169 good_area:
170     info.si_code = SEGV_ACCERR;
171     write = 0;
172     switch (error_code & 3) {
173         default: /* 3: write, present */
174 #ifdef TEST_VERIFY_AREA
175             if (regs->cs == KERNEL_CS)
176                 printk("WP fault at %08lx\n", regs->eip);
177 #endif
178             /* fall through */
179         case 2:     /* write, not present */
180             if (!(vma->vm_flags & VM_WRITE))
181                 goto bad_area;
182             write++;
183             break;
184         case 1:     /* read, present */
185             goto bad_area;
186         case 0:     /* read, not present */
187             if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
188                 goto bad_area;
189     }
190 
191     /*
192      * If for any reason at all we couldn't handle the fault,
193      * make sure we exit gracefully rather than endlessly redo
194      * the fault.
195      */
196     switch (handle_mm_fault(mm, vma, address, write)) {
197     case 1:
198         tsk->min_flt++;
199         break;
200     case 2:
201         tsk->maj_flt++;
202         break;
203     case 0:
204         goto do_sigbus;
205     default:
206         goto out_of_memory;
207     }
```

在这里的 switch 语句中，内核根据由中断响应机制传过来的 error_code 来进一步确定映射失败的原因并采取相应的对策（error_code 最低三位的定义已经在前节中列出）。就现在这个情景而言，bit0 为 0，表示没有物理页面，而 bit1 为 1 表示写操作。所以，最低两位的值为 2。既然是写操作，当然要检查相应的区间是否允许写入，而堆栈段是允许写入的。于是，就到达 196 行，调用虚存管理 handle_mm_fault()了。该函数定义于 mm/memory.c 中：

==================== mm/memory.c 1189 1208 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[do_page_fault()>handle_mm_fault()]

```
/*
 * By the time we get here, we already hold the mm semaphore
 */
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma,
	unsigned long address, int write_access)
{
	int ret = -1;
	pgd_t *pgd;
	pmd_t *pmd;

	pgd = pgd_offset(mm, address);
	pmd = pmd_alloc(pgd, address);

	if (pmd) {
		pte_t * pte = pte_alloc(pmd, address);
		if (pte)
			ret = handle_pte_fault(mm, vma, address, write_access, pte);
	}
	return ret;
}
```

根据给定的地址和代表着具体虚存空间的 mm_struct 数据结构，由宏操作 pgd_offset()计算出指向该地址所属页面目录项的指针。这是在 include/asm-i386/pgtable.h 中定义的：

==================== include/asm-i386/pgtable.h 311 312 ====================

```
/* to find an entry in a page-table-directory. */
#define pgd_index(address) ((address >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))
  ......
```

==================== include/asm-i386/pgtable.h 316 316 ====================

```
#define pgd_offset(mm, address) ((mm)->pgd+pgd_index(address))
```

至于下面的 pmd_alloc()，本来是应该分配（或者找到）一个中间目录项的。由于 i386 只使用两层映射，所以在 include/asm_i386/pgtable_2level.h 中将其定义为“ return (pmd_t *)pgd; ”。也就是说，在 i386 CPU 中，把具体的目录项当成一个只含一个表项（表的大小为 1）的中间目录。所以，对于 i386 CPU 而言，pmd_alloc()是绝不会失败的，所以这里的 pmd 不可能为 0。读者不妨顺着线性地址的映射过程想想，接下来需要做些什么？页面目录总是在的，相应的目录项也许已经指向一个页面表，此时需要根据给定的地址在表中找到相应的页面表项。或者，目录项也可能还是空的，那样的话就需要先分配一个页面表，再在页面表中找到相应的表项。这样，才可以为下面分配物理内存页面并建立映射做好准备。这是通过 pte_alloc()完成的，其代码在 include/asm-i386/pgalloc.h 中：

==================== include/asm-i386/pgalloc.h 120 141 ====================

[do_page_fault()>handle_mm_fault()>pte_alloc()]

```
extern inline pte_t * pte_alloc(pmd_t * pmd, unsigned long address)
{
	address = (address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	if (pmd_none(*pmd))
		goto getnew;
	if (pmd_bad(*pmd))
		goto fix;
	return (pte_t *)pmd_page(*pmd) + address;
getnew:
{
	unsigned long page = (unsigned long) get_pte_fast();

	if (!page)
		return get_pte_slow(pmd, address);
	set_pmd(pmd, __pmd(_PAGE_TABLE + __pa(page)));
	return (pte_t *)page + address;
}
fix:
	__handle_bad_pmd(pmd);
	return NULL;
}
```

先将给定的地址转换成其所属页面表中的下标。在我们这个情景中，假定指针 pmd 所指向的目录项为空，所以需要转到标号 get_new()处分配一个页面表。一个页面表所占的空间恰好是一个物理页面。内核中对页面表的分配作了些优化。当释放一个页面表时，内核将释放的页面表先保存在一个缓冲池中，而先不将其物理内存页面释放。只有在缓冲池已满的情况下才真的将页面表所占的物理内存页面释放。这样，在要分配一个页面表时，就可以先看一下缓冲池，这就是 get_pte_fast()。要是缓冲池已经空了，那就只好通过 get_pte_kernel_slow()来分配了。读者也许会想，分配一个物理内存页面用作页面表就那么麻烦吗，为什么是“slow”呢？回答是有时候可能会很慢。只要想一下物理内存页面有可能已经用完，需要把内存中已经占用的页面交换到磁盘上去，就可以明白了。分配到一个页面表以后，就通过 set_pmd()中将其起始地址连同一些属性标志位一起写入中间目录项 pmd 中，而对 i386 却实际上写入到了页面目录项 pgd 中。这样，映射所需的“基础设施”都已经齐全了，但页面表项 pte 还是空的。剩下的就是物理内存页面本身了，那是由 handle_pte_fault()完成的。该函数定义于 mm/memory.c 内：

==================== mm/memory.c 1135 1187 ====================
[do_page_fault()>handle_mm_fault()>handle_pte_fault()]

```
/*
 * These routines also need to handle stuff like marking pages dirty
 * and/or accessed for architectures that don't do it in hardware (most
 * RISC architectures).  The early dirtying is also good on the i386.
 *
 * There is also a hook called "update_mmu_cache()" that architectures
 * with external mmu caches can use to update those (ie the Sparc or
 * PowerPC hashed page tables that act as extended TLBs).
 *
 * Note the "page_table_lock". It is to protect against kswapd removing
 * pages from under us. Note that kswapd only ever _removes_ pages, never
 * adds them. As such, once we have noticed that the page is not present,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * we can drop the lock early.
 *
 * The adding of pages is protected by the MM semaphore (which we hold),
 * so we don't need to worry about a page being suddenly been added into
 * our VM.
 */
static inline int handle_pte_fault(struct mm_struct *mm,
     struct vm_area_struct * vma, unsigned long address,
     int write_access, pte_t * pte)
{
     pte_t entry;

     /*
      * We need the page table lock to synchronize with kswapd
      * and the SMP-safe atomic PTE updates.
      */
     spin_lock(&mm->page_table_lock);
     entry = *pte;
     if (!pte_present(entry)) {
          /*
           * If it truly wasn't present, we know that kswapd
           * and the PTE updates will not touch it later. So
           * drop the lock.
           */
          spin_unlock(&mm->page_table_lock);
          if (pte_none(entry))
               return do_no_page(mm, vma, address, write_access, pte);
          return do_swap_page(mm, vma, address, pte, pte_to_swp_entry(entry), write_access);
     }

     if (write_access) {
          if (!pte_write(entry))
               return do_wp_page(mm, vma, address, pte, entry);

          entry = pte_mkdirty(entry);
     }
     entry = pte_mkyoung(entry);
     establish_pte(vma, address, pte, entry);
     spin_unlock(&mm->page_table_lock);
     return 1;
}
```

在我们这个情景里，不管页面表是新分配的还是原来就有的，相应的页面表项却一定是空的。这样，程序一开头的 if 语句的条件一定能满足，因为 pte_present()测试一个表项所映射的页面是否在内存中，而我们的物理内存页面还没有分配。进一步，pte_none()所测试的条件也一定能满足，因为它测试一个表项是否为空。所以，就必定会进入 do_no_page()（否则就是 do_swap_page()）。顺便讲一下，如果 pte_present()的测试结果是该表项所映射的页面确实在内存中，那么问题一定出在访问权限，或者根本就没有问题了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

函数 do_no_page()也是在 mm/memory.c 中定义的。这里先简要地介绍一下，然后再来看代码。

以前我们曾经提起过，在虚存区间结构 vm_area_struct 中有个指针 vm_ops，指向一个 vm_operations_struct 数据结构。这个数据结构实际上是一个函数跳转表，结构中通常是一些与文件操作有关的函数指针。其中有一个函数指针就是用于物理内存页面的分配。物理内存页面的分配为什么与文件操作有关呢？因为这对于可能的文件共享是很有意义的。当多个进程将同一个文件映射到各自的虚存空间中时，内存中通常只要保存一份物理页面就可以了。只有当一个进程需要写入该文件时才有必要另外复制一份独立的副本，称为“copy on write”或者 COW。关于 COW 我们在进程一章中讲到 fork()还要作较为详细的介绍。这样，当通过 mmap()将一块虚存区间跟一个已打开文件（包括设备）建立起映射后，就可以通过对这些函数的调用将对内存的操作转化成对文件的操作，或者进行一些必要的对文件的附加操作。另一方面，物理页面的盘区交换显然也是跟文件操作有关的。所以，为特定的虚存空间预先指定一些特定的操作常常是很有必要的。于是，如果已经预先为一个虚存区间 vma 指定了分配物理内存页面的操作的话，那就是 vma->vm_ops->nopage()。但是，vma->vm_ops 和 vma->vm_ops->nopage 都有可能是空，那就表示没有为之指定具体的 nopage()操作，或者根本就没有配备一个 vm_operations_struct 结构。当没有指定的 nopage()操作时，内核就调用一个函数 do_anonymous_page()来分配物理内存页面。

现在来看看 do_no_page()的开头几行：

==================== mm/memory.c 1080 1098 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()]

```
1080 /*
1081  * do_no_page() tries to create a new page mapping. It aggressively
1082  * tries to share with existing pages, but makes a separate copy if
1083  * the "write_access" parameter is true in order to avoid the next
1084  * page fault.
1085  *
1086  * As this is called only for pages that do not currently exist, we
1087  * do not need to flush old virtual caches or the TLB.
1088  *
1089  * This is called with the MM semaphore held.
1090  */
1091 static int do_no_page(struct mm_struct * mm, struct vm_area_struct * vma,
1092      unsigned long address, int write_access, pte_t *page_table)
1093 {
1094      struct page * new_page;
1095      pte_t entry;
1096
1097      if (!vma->vm_ops || !vma->vm_ops->nopage)
1098          return do_anonymous_page(mm, vma, page_table, write_access, address);
   ......
```

==================== mm/memory.c 1133 1133 ====================

```
1133 }
```

对于我们这个情景来说，所涉及的虚存区间是供堆栈用的，跟文件系统或页面共享没有什么关系，不会有指定的 nopage()操作，所以进入 do_anonymous_page()。

==================== mm/memory.c 1058 1078 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()>do_anonymous_page()]

```
/*
 * This only needs the MM semaphore
 */
static int do_anonymous_page(struct mm_struct * mm, struct vm_area_struct * vma, pte_t *page_table, int write_access, unsigned long addr)
{
	struct page *page = NULL;
	pte_t entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
	if (write_access) {
		page = alloc_page(GFP_HIGHUSER);
		if (!page)
			return -1;
		clear_user_highpage(page, addr);
		entry = pte_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)));
		mm->rss++;
		flush_page_to_ram(page);
	}
	set_pte(page_table, entry);
	/* No need to invalidate - it was non-present before */
	update_mmu_cache(vma, addr, entry);
	return 1;/* Minor fault */
}
```

首先我们注意到，如果引起页面异常的是一次读操作，那么由 mk_pte()构筑的映射表项要通过 pte_wrprotect()加以修正；而如果是写操作（参数 write_access 为非 0），则通过 pte_mkwrite()加以修正。这二者有什么不同呢？见 include/asm-i386/pgtable.h：

==================== include/asm-i386/pgtable.h 277 277 ====================

```
static inline pte_t pte_wrprotect(pte_t pte)  { (pte).pte_low &= ~_PAGE_RW; return pte; }
```

==================== include/asm-i386/pgtable.h 271 271 ====================

```
static inline int pte_write(pte_t pte)        { return (pte).pte_low & _PAGE_RW; }
```

对比一下，就可看出，在 pte_wrprotect()中，把_PAGE_RW 标志位设成 0，表示这个物理页面只允许读；而在 pte_write()却把这个标志位设成 1。同时，对于读操作，所映射的物理页面总是 ZERO_PAGE，这个页面是在 include/asm-i386/pgtable.h 中定义的：

==================== include/asm-i386/pgtable.h 91 96 ====================

```
/*
 * ZERO_PAGE is a global shared page that is always zero: used
 * for zero-mapped memory areas etc..
 */
extern unsigned long empty_zero_page[1024];
#define ZERO_PAGE(vaddr) (virt_to_page(empty_zero_page))
```

就是说，只要是“只读”（也就是写保护）的页面，开始时都一律映射到同一个物理内存页面 empty_zero_page，而不管其虚拟地址是什么。实际上，这个页面的内容为全 0，所以映射之初若从该

PDF-XCHANGE Click to buy NOW! www.docu-track.com

页面读出就读得 0。只有可写的页面，才通过 alloc_page()为其分配独立的物理内存。在我们这个情景里，所需要的页面是在堆栈区，并且是由于写操作才引起异常的，所以要通过 alloc_page()为其分配一个物理内存页面，并将分配到的物理页面连同所有的状态及标志位（见程序 1115 行），一起通过 set_pte()设置进指针 page_table 所指的页面表项。至此，从虚存页面到物理内存页面的映射终于建立了。这里的 update_mmu_cache()对 i386 CPU 是个空函数（见 include/asm-i386/pgtable.h），因为 i386 的 MMU（内存管理单元）是实现在 CPU 内部，而并没有独有的 MMU。

映射既已建立，下面就是逐层返回了。由于映射成功，各个层次中的返回值都是 1，直至 do_page_fault()。在函数 do_page_fault()中，还要处理一个与 VM86 模式以及 VGA 的图像存储区有关的特殊情况，但是那与我们这个情景已经没有关系了：

==================== arch/i386/mm/fault.c 209 218 ====================

[do_page_fault()]

```
	/*
	 * Did it hit the DOS screen memory VA from vm86 mode?
	 */
	if (regs->eflags & VM_MASK) {
		unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
		if (bit < 32)
			tsk->thread.screen_bitmap |= 1 << bit;
	}
	up(&mm->mmap_sem);
	return;
```

最后，特别要指出，当 CPU 从一次页面错异常处理返回到用户空间时，将会先重新执行因映射失败而中途夭折的那条指令，然后才继续往下执行，这是异常处理的特殊性。学过有关课程的读者都知道，中断以及自陷（trap 指令）发生时，CPU 都会将下一条指令，也就是接下去本来应该执行的指令的地址压入堆栈作为中断服务的返回地址。但是异常却不同。当异常发生时，CPU 将因无法完成（例如除以 0，映射失败，等等）而夭折的指令本身的地址（而不是下一条指令的地址）压入堆栈。这样，就可以在从异常处理返回时完成未竟的事业。这个特殊性是在 CPU 的内部电路中实现的，而不需由软件干预。从这个意义上讲，所谓“缺页中断”是不对的，应该叫“缺页异常”才对。在我们这个情景中，当初是因为在一条指令中要压栈，但是越出了已经为堆栈区分配的空间而引起的。那条指令在当时已经中途夭折了，并没有产生什么效果（例如堆栈指针%esp 还是指向原来的位置）。现在，从异常处理返回以后，堆栈区已经扩展了，再重新执行一遍以前夭折的那条压栈指令，然后就可以继续往下执行了。对于用户程序来说，这整个过程都是“透明”的，就像什么事也没有发生过，而堆栈区间就仿佛从一开始就已经分配好了足够大的空间一样。

2.6 物理页面的使用和周转

除 CPU 之外，对于像 Linux 这样的现代操作系统来说，物理存储页面可以说是最基本、最重要的资源了。物理存储页面在系统中的使用和周转就好像资金在企业中的使用和周转一样重要。因此，读者对此最好能有更多一些了解。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

首先要澄清本书中使用的几个术语。“虚存页面”，是指在虚拟地址空间中一个固定大小，边界与页面大小（4KB）对齐的区间及其内容。虚存页面最终要落实到，或者说要映射到某种物理存储介质上，那就是“物理页面”。根据具体介质的不同，一个物理页面可以在内存中，也可以在磁盘上。为了区分这两种情况，本书将分别称之为“（物理）内存页面”和“盘上（物理）页面”。此外，在某项外部设备上，例如在网络接口卡上，用来存储一个页面内容的那部分介质，也称为一个物理页面。所以，当我们在谈及物理内存页面的分配和释放的时候，指的仅是物理介质，而在谈及页面的换入和换出时则指的是其内容。读者，特别是非计算机专业的读者，一定要清楚并记住这一点。

如前所述，每个进程的虚存空间是很大的（用户空间为 3GB）。不过，每个进程实际上使用的空间则要小得多，一般不会超过几个 MB。特别地，传统的 Linux（以及 Unix）可执行程序通常都是比较小的，例如几十 KB 或一二百 KB。可是，当系统中有几百个、上千个进程同时存在的时候，对存储空间的需求总量就很大了。在这样的情况下，要为系统配备足够的内存就很难。所以，在计算机技术的发展史上很早就有了把内存的内容与一个专用的磁盘空间“交换”的技术，即把暂时不用的信息（内容）存放到磁盘上，为其它急用的信息腾出空间，到需要时再从磁盘上读进来的技术。早期的盘区交换技术是建立在段式存储管理的基础上的，当一个进程暂不运行的时候就可以把它（代码段和数据段等）交换出去（把其它进程换进来，故曰“交换”），到调度这个进程运行时再交换回来。显然，这样的盘区交换是很粗糙的，对系统性能的影响也比较大，所以后来发展起了建立在页式存储管理基础上的“按需页面交换”技术。

在计算机技术中，时间和空间是一对矛盾，常常需要在二者之间折中权衡，有时候是以空间换时间，有时候是以时间换空间。而页面的交换，则是典型的以时间换空间。必须指出，这只是不得已而为之。特别是在有实时要求的系统中，是不宜采用页面交换的，因为它使程序的执行在时间上有了较大的不确定性。因此，Linux 提供了用来开启和关闭页面交换机制的系统调用，不过我们在本章的叙述中假定它是打开的。

在介绍页面周转的策略之前，先要对物理页面、特别是磁盘页面的抽象描述作一个简要说明。

前面已经简略地介绍过，为了方便（物理）内存页面的管理，每一个内存页面都对应一个 page 数据结构。每一个物理内存页面之有 page 数据结构（以及每个进程之有其 task_struct 结构），就好像每个人之有“户口”或者“档案”一样。一个物理上存在的人，如果没有户口，从管理的角度来说便是不存在的。同样，一个物理上存在的内存页面，如果没有一个相应的 page 结构，就根本不会被系统“看到”。在系统的初始化阶段，内核根据检测到的物理内存的大小，为每一个页面都建立一个 page 结构，形成一个 page 结构的数组，并使一个全局量 mem_map 指向这个数组。同时，又按需要将这些页面拼合成物理地址连续的许多内存页面“块”，再根据块的大小建立起若干“管理区”（zone），而在每个管理区中则设置一个空闲块队列，以便物理内存页面的分配使用。这一些，读者已经在前面看到过了。

与此类似，交换设备（通常是磁盘，也可以是普通文件）的每个物理页面也要在内存中有个相应的数据结构（或者说“户口”），不过那要简单得多，实际上只是一个计数，表示该页面是否已被分配使用，以及有几个用户在共享这个页面。对盘上页面的管理是按文件或磁盘设备来进行的。内核中定义了一个 swap_info_struct 数据结构，用以描述和管理用于页面交换的文件或设备。它的定义包含在 include/linux/swap.h 中：

==================== include/linux/swap.h 49 64 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
struct swap_info_struct {
        unsigned int flags;
        kdev_t swap_device;
        spinlock_t sdev_lock;
        struct dentry * swap_file;
        struct vfsmount *swap_vfsmnt;
        unsigned short * swap_map;
        unsigned int lowest_bit;
        unsigned int highest_bit;
        unsigned int cluster_next;
        unsigned int cluster_nr;
        int prio;           /* swap priority */
        int pages;
        unsigned long max;
        int next;           /* next entry on swap list */
};
```

其中的指针 swap_map 指向一个数组，该数组中的每一个无符号短整数即代表盘上（或普通文件中）的一个物理页面，而数组的下标则决定了该页面在盘上或文件中的位置。数组的大小取决于 pages，它表示该页面交换设备或文件的大小。设备上（或文件中，设备也是一种文件，下同）的第一个页面，也即 swap_map[0]所代表的那个页面是不用于页面交换的，它包含了该设备或文件自身的一些信息以及一个表明哪些页面可供使用的位图。这些信息最初是在把该设备格式化成页面交换区时设置的。根据不同的页面交换区格式（以及版本），还有一些其它的页面也不供页面交换使用。这些页面都集中在开头和结尾两个地方，所以 swap_info_struct 结构中的 lowest_bit 和 highest_bit 就说明文件中从什么地方开始到什么地方为止是供页面交换使用的。另一个字段 max 则表示该设备或文件中最大的页面号，也就是设备或文件的物理大小。

由于存储介质是转动的磁盘，将地址连续的页面存储在连续的磁盘扇区中不见得是最有效的方法，所以在分配盘上页面空间时尽可能按集群（cluster）方式进行，而字段 cluster_next 和 cluster_nr 就是为此而设置的。

Linux 内核允许使用多个页面交换设备（或文件），所以在内核中建立了一个 swap_info_struct 结构的阵列（数组）swap_info，这是在 mm/swapfile.c 中定义的：

==================== mm/swapfile.c 25 25 ====================

```
struct swap_info_struct swap_info[MAX_SWAPFILES];
```

同时，还设立了一个队列 swap_list，将各个可以分配物理页面的磁盘设备或文件的 swap_info_struct 结构按优先级高低链接在一起：

==================== mm/swapfile.c 23 23 ====================

```
struct swap_list_t swap_list = {-1, -1};
```

这里的 swap_list_t 数据结构是在 include/linux/swap.h 中定义的：

==================== include/linux/swap.h 153 156 ====================

```
struct swap_list_t {
        int head; /* head of priority-ordered swapfile list */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int next;/* swapfile to be used next */
};
```

开始时队列为空，所以 head 和 next 均为-1。当系统调用 swap_on()指定一个文件用于页面交换时，就将该文件的 swap_info_struct 结构链入队列中。

就像通过 pte_t 数据结构（页面表项）将物理内存页面与虚存页面建立联系一样，盘上页面也有这么一个 swp_entry_t 数据结构，这是在 include/linux/shmem_fs.h 中定义的：

==================== include/linux/shmem_fs.h 8 18 ====================

```
/*
 * A swap entry has to fit into a "unsigned long", as
 * the entry is hidden in the "index" field of the
 * swapper address space.
 *
 * We have to move it here, since not every user of fs.h is including
 * mm.h, but m.h is including fs.h via sched .h :-/
 */
typedef struct {
    unsigned long val;
} swp_entry_t;
```

可见，一个 swp_entry_t 结构实际上只是一个 32 位无符号整数。但是，这个 32 位整数实际上分成三个部分，见图 2.7。

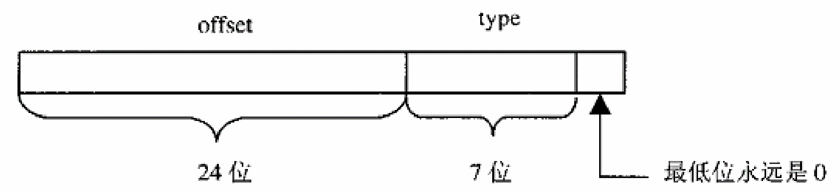


图 2.7　页面交换项结构示意图

文件 include/asm-i386/pgtable.h 中还为 type 和 offset 两个位段的访问以及与 pte_t 结构之间的关系，定义了几个宏操作：

==================== include/asm-i386/pgtable.h 336 341 ====================

```
/* Encode and de-code a swap entry */
#define SWP_TYPE(x)              (((x).val >> 1) & 0x3f)
#define SWP_OFFSET(x)            ((x).val >> 8)
#define SWP_ENTRY(type, offset)       ((swp_entry_t) { ((type) << 1) | ((offset) << 8) })
#define pte_to_swp_entry(pte)         ((swp_entry_t) { (pte).pte_low })
#define swp_entry_to_pte(x)      ((pte_t) { (x).val })
```

这里 offset 表示页面在一个磁盘设备或文件中的位置，也就是文件中的逻辑页面号；而 type 则是指该页面在哪一个文件中，是个序号。这个位段的命名很容易引起读者的误解，明明是指页面交换设备或文件的序号（一共可以容纳 127 个这样的文件，但实际上则视系统的配置而定，远小于 127），为什么却称之为 type 呢？估计这是从 pte_t 结构中过来的。读者可能记得，pte_t 实际上也是一个 32 位无

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

符号整数，其中最高的 20 位为物理页面起始地址的高 20 位（物理页面起始地址的低 12 位永远是 0，因为页面都是 4KB 边界对齐的），而与这 7 位相对应的则都是些表示页面各种性质的标志位，如 R/W，U/S，等等，所以称之为 type 位段。而 swp_entry_t 与 pte_t 两种数据结构大小相同，关系非常密切。当一个页面在内存中时，页面表中的表项 pte_t 的最低位 P 标志为 1，表示页面在内存中，而其余各位指明物理内存页面的地址及页面的属性。而当一个页面在磁盘上时，则相应的页面表项不再指向一个物理内存页面，而是变成了一个 swp_entry_t "表项"，指示着这个页面的去向。由于此时其最低位为 0，表示页面不在内存，所以 CPU 中的 MMU 单元对其余各位都忽略不顾，而留待系统软件自己来加以解释。在 Linux 内核中，就用它来惟一地确定一个页面在盘上的位置，包括在哪一个文件或设备，以及页面在此文件中的相对位置。

所以，当页面在内存时，页面表中的相应表项确定了地址的映射关系；而当页面不在内存时，则指明了物理页面的去向和所在。读者在阅读内核的源程序时，不妨将 SWP_TYPE(entry)想像成 SWP_FILE(entry)。

下面转入本节标题所说对物理页面周转的介绍。我们还是通过一些函数的代码来帮助读者理解。

先介绍一下用来释放一个磁盘页面的函数__swap_free()。通过这个函数的阅读，读者可以加深对上面这一段说明的理解。此函数的代码在文件 mm/swapfile.c 中。而分配磁盘页面的函数__get_swap_page()也在同一文件中，读者不妨自行阅读。

先来看__swap_free()的开头几行：

==================== mm/swapfile.c 141 158 ====================

```
141  /*
142   * Caller has made sure that the swapdevice corresponding to entry
143   * is still around or has not been recycled.
144   */
145  void __swap_free(swp_entry_t entry, unsigned short count)
146  {
147       struct swap_info_struct * p;
148       unsigned long offset, type;
149
150       if (!entry.val)
151            goto out;
152
153       type = SWP_TYPE(entry);
154       if (type >= nr_swapfiles)
155            goto bad_nofile;
156       p = & swap_info[type];
157       if (!(p->flags & SWP_USED))
158            goto bad_device;
```

如果 entry.val 为 0，就显然不需要做任何事，因为在任何页面交换设备或文件中页面 0 是不用于页面交换的。接着，如前所述，SWAP_TYPE 所返回的实际上是页面交换设备的序号，即其 swap_info_struct 结构在 swap_info[]数组中的下标。所以 156 行以此为下标从 swap_info[]中取得具体文件的 swap_info_struct 结构。文件找到以后，下面就来看具体的页面了：

==================== mm/swapfile.c 159 182 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	offset = SWP_OFFSET(entry);
	if (offset >= p->max)
		goto bad_offset;
	if (!p->swap_map[offset])
		goto bad_free;
	swap_list_lock();
	if (p->prio > swap_info[swap_list.next].prio)
		swap_list.next = type;
	swap_device_lock(p);
	if (p->swap_map[offset] < SWAP_MAP_MAX) {
		if (p->swap_map[offset] < count)
			goto bad_count;
		if (!(p->swap_map[offset] -= count)) {
			if (offset < p->lowest_bit)
				p->lowest_bit = offset;
			if (offset > p->highest_bit)
				p->highest_bit = offset;
			nr_swap_pages++;
		}
	}
	swap_device_unlock(p);
	swap_list_unlock();
out:
	return;
```

如前所述，offset 是页面在文件中的位置，当然不能大于文件本身所提供的最大值。而 p->swap_map[offset]是该页面的分配（和使用）计数，如为 0 就表明尚未分配。同时，分配计数也不应大于 SWAP_MAP_MAX。函数的调用参数 count 表示有几个使用者释放该页面，所以从计数中减去 count。当计数达到 0 时，这个页面就真正变成空闲了。此时，如果页面落在当前可供分配的范围之外，就要相应地调整这个范围的边界 lowest_bit 或 highest_bit，同时，可供分配的盘上页面的数量 nr_swap_pages 也增加了。值得注意的是，释放磁盘页面的操作实际上并不涉及磁盘操作，而只是在内存中“账面”上的操作，表示磁盘上那个页面的内容已经作废。所以，花费的代价是极小的。

知道了内核怎样管理内存页面和盘上页面以后，就可以来看看内存页面的周转了。当一个内存页面空闲，也就是留在某一个空闲页面管理区的空闲队列中时，其 page 结构中的计数 count 为 0，而在分配页面时将其设置成 1。这是在函数 rmqueue()中通过 set_page_count()设置的，我们在前面已经看到过。

所谓内存页面的周转有两方面的意思。其一是页面的分配、使用和回收，并不一定涉及页面的盘区交换。其二才是盘区交换，而交换的目的最终也是页面的回收。并非所有的内存页面都是可以交换出去的。事实上，只有映射到用户空间的页面才会被换出，而内核，即系统空间的页面则不在此列。这里要说明一下，在内核中可以访问所有的物理页面，换言之所有的物理页面在系统空间中都是有映射的。所谓“用户空间的页面”，是指在至少一个进程的用户空间中有映射的页面，反之则为（只能由）内核使用的页面。

按页面的内容和性质，用户空间的页面有下面几种：

- 普通的用户空间页面，包括进程的代码段、数据段、堆栈段，以及动态分配的“存储堆”。其

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

中有些页面从用户程序即进程的角度看是静态的（如代码段），但从系统的角度看仍是动态分配的。

- 通过系统调用 mmap()映射到用户空间的已打开文件的内容。
- 进程间的共享内存区。

这些页面既涉及分配、使用和回收，也涉及页面的换出 / 换入。

凡是映射到系统空间的页面都不会被换出，但还是可以按使用和周转的不同而大致上分成几类。首先，内核代码和内核中全局量所占的内存页面既不需要经过分配也不会被释放，这部分空间是静态的。（相比之下，进程的代码段和全局量都在用户空间，所占的内存页面都是动态的，使用前要经过分配，最后都会被释放，并且中途可能被换出而回收后另行分配）

除此之外，内核中使用的内存页面也要经过动态分配，但永远都保留在内存中，不会被交换出去。此类常驻内存的页面根据其内容的性质可以分成两类。

一类是一旦使用完毕便无保存的价值，所以立即便可释放、回收。这类页面的周转很简单，就是空闲→（分配）→使用→（释放）→空闲。这种用途的内核页面大致上有这样一些：

- 内核中通过 kmalloc()或 vmalloc()分配、用作某些临时性使用和为管理目的而设的数据结构，如 vma_area_struct 数据结构等等。这些数据结构一旦使用完毕便无保存价值，所以立即便可释放。不过由于一个页面中往往有多个同种数据结构，所以要到整个页面都空闲时才能把页面释放。
- 内核中通过 alloc_page()分配，用作某此临时性使用和为管理目的的内存页面，如每个进程的系统堆栈所在的两个页面，以及从系统空间复制参数时使用的页面等等。这些页面也是一旦使用完毕便无保存的价值，所以立即便可释放。

另一类是虽然使用完毕了，但是其内容仍有保存的价消。只要条件允许，把这些页面“养起来 ”也许可以提高以后的操作效率。这类页面（或数据结构）在“释放”之后要放入一个 LRU 队列，经过一段时间的缓冲让其“老化”；如果在此期间忽然又要用到其内容了，便直接将页面连内容分配给“用户”；否则便继续老化，直到条件不再允许时才加以回收。这种用途的内核页面大致上有下面这些：

- 在文件系统操作中用来缓冲存储一些文件目录结构 dentry 的空间。
- 在文件系统操作中用来缓冲存储一些 inode 结构的空间。
- 用于文件系统读 / 写操作的缓冲区。

这些页面的内容是从文件系统中直接读入或经过综合取得的，释放后立即回收另作它用也并无不可，但是那样以后要用时就又要付出代价了。

相比之下，页面交换是最复杂的，所以我们将花较大的篇幅来介绍。

显然，最简单的页面交换策略就是：每当缺页异常时便分配一个内存页面，并把在磁盘上的页面读入到分配得到的内存页面中。如果没有空闲页面可供分配，就设法将一个或几个内存页面换出到磁盘上，从而腾出一些内存页面来。但起，这种完全消极的页面交换策略有个缺点，就是页面的交换总是“临阵磨枪”，发生在系统忙碌的时候而没有调度的余地。比较积极的办法是定期地，最好是在系统相对空闲时，挑选一些页面预先换出而腾出一些内存页面，从而在系统中维持一定的空闲页面供应量，使得在缺页中断发生时总是有空闲内存页面可供分配。至于挑选的准则，一般都是 LRU，即挑选“最近最少用到”的页面。但是，这种积极的页面交换策略实行起来也有问题，因为实际上并不存在一种方法可以准确地预测对页面的访问。所以，完全有可能发生这样的现象，就是一个页面已经好久没有

PDF-XCHANGE Click to buy NOW! www.docu-track.com

受到访问了，但是刚把它换出到磁盘上，却又有访问了，于是只好又赶快把它换进来。在最坏的情况下，有可能整个系统的处理能力都被这样的换入 / 换出所饱和，而实际上根本不能进行有效的运算和操作。有人把此种现象称为（页面的）“抖动”。

为了防止这种情况的发生，可以将页面的换出和内存页面的释放分成两步来做。当系统挑选出若干内存页面准备换出时，将这些页面的内容写入相应的磁盘页面中，并且将相应页面表项的内容改成指向盘上页面（P 标志位为 0，表示页面不在内存），但是所占据的内存页面却并不立即释放，而是将其 page 结构留在一个“暂存”（cache）队列（或称缓冲队列）中，只是使其从“活跃状态”转入了“不活跃状态”，就像军人从“现役”转入了“预备役”。至于内存页面的“退役”，即最后释放，则推迟到以后有条件地进行。这样，如果在一个页面被换出以后立即又受到访问而发生缺页异常，就可以从物理页面的暂存队列中找回相应的页面，再次为之建立映射。由于此页面尚未释放，还保留着其原来的内容，就不需要从盘上读入了。反之，如果经过一段时间以后，一个不活跃的内存页面，即还留在暂存队列却已不再有（用户空间）映射的页面，还是没有受到访问，那就到了最后退役的时候了。如果留在暂存队列中的页面又受到访问，确切地说是发生了以此页面为目标的页面异常，那么只要恢复这个页面的映射并使其脱离暂存队列就可以了，此时该页面又回到了活跃状态。

这种策略显然可以减小抖动的可能，并且减少系统在页面交换上的花费。可是，如果更深入地考察这个问题，就可以看出其实还可以改进。首先，在准备换出一个页面时并不一定要把它的内容写入磁盘。如果自从最近一次换入该页面以后从未写过这个页面，那么这个页面是“干净”的，也就是与盘上页面的内容相一致，这样的页面当然不用写出去。其次，即使是“脏”的页面，也不必立刻就写出去，而可以先从页面映射表断开，经过一段时间的“冷却”或“老化”后再写出去，从而变成“干净”页面。至于“干净”页面，则还可以继续缓冲到真有必要时才加以回收，因为回收一个“干净 ”页面的花费是很小的。

综上所述，物理内存页面换入 / 换出的周转要点如下：

(1) 空闲。页面的 page 数据结构通过其队列头结构 list 链入某个页面管理区（zone）的空闲区队列 free_area。页面的使用计数 count 为 0。

(2) 分配。通过函数__alloc_pages()或__get_free_page()从某个空闲队列中分配内存页面，并将所分配页面的使用计数 count 置成 1，其 page 数据结构的队列头 list 结构则变成空闲。

(3) 活跃状态。页面的 page 数据结构通过其队列头结构 lru 链入活跃页面队列 active_list，并且至少有一个进程的（用户空间）页面表项指向该页面。每当为页面建立或恢复映射时，都使页面的使用计数 count 加 1。

(4) 不活跃状态（脏）。页面的 page 数据结构通过其队列头结构 lru 链入不活跃“脏”页面队列 inactive_dirty_list，但是原则上不再有任何进程的页面表项指向该页面。每当断开页面的映射时都使页面的使用计数 count 减 1。

(5) 将不活跃“脏”页面的内容写入交换设备，并将页面的 page 数据结构从不活跃“脏”页面队列 inactive_dirty_list 转移到某个不活跃“干净”页面队列中。

(6) 不活跃状态（干净）。页面的 page 数据结构通过其队列头结构 lru 链入某个不活跃“干净”页面队列，每个页面管理区都有一个不活跃“干净”页面队列 inactive_clean_list。

(7) 如果在转入不活跃状态以后的一段时间内页面受到访问，则又转入活跃状态并恢复映射。

(8) 当有需要时，就从“干净”页面队列中回收页面，或退回到空闲队列中，或直接另行分配。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

当然，实际的实现还要更复杂一些。

为了实现这种策略，在 page 数据结构中设置了所需的各种成分，并在内核中设置了全局性的 active_list 和 inactive_dirty_list 两个 LRU 队列，还在每个页面管理区中设置了一个 inactive_clean_list。根据页面的 page 结构在这些 LRU 队列中的位置，就可以知道这个页面转入不活跃状态后时间的长短。为回收页面提供参考。同时，还通过一个全局的 address_space 数据结构 swapper_space，把所有可交换内存页面管理起来，每个可交换内存页面的 page 数据结构都通过其队列头结构 list 链入其中的一个队列。此外，为加快在暂存队列中的搜索，又设置了一个杂凑表 page_hash_table。

让我们来看看内核是怎样将一个内存页面链入这些队列的。内核在为某个需要换入的页面分配了一个空闲内存页面以后，就通过 add_to_swap_cache()将其 page 结构链入相应的队列，这个函数的代码在 mm/swap_state.c 中：

==================== mm/swap_state.c 54 70 ====================

```
void add_to_swap_cache(struct page *page, swp_entry_t entry)
{
	unsigned long flags;

#ifdef SWAP_CACHE_INFO
	swap_cache_add_total++;
#endif
	if (!PageLocked(page))
		BUG();
	if (PageTestandSetSwapCache(page))
		BUG();
	if (page->mapping)
		BUG();
	flags = page->flags & ~((1 << PG_error) | (1 << PG_arch_1));
	page->flags = flags | (1 << PG_uptodate);
	add_to_page_cache_locked(page, &swapper_space, entry.val);
}
```

在调用这个函数前要先将页面锁住，以免受到干扰。因为是刚分配的空闲页面，其 PG_swap_cache 标志位必须为 0，指针 mapping 也必须为 0。同时，页面的内容是刚从交换设备读入的，当然与盘上页面一致，所以把 PG_uptodate 标志位设成 1。函数__add_to_page_cache()的定义见 mm/filemap.c：

==================== mm/filemap.c 476 494 ====================

```
/*
 * Add a page to the inode page cache.
 *
 * The caller must have locked the page and
 * set all the page flags correctly..
 */
void add_to_page_cache_locked(struct page * page, struct address_space *mapping, unsigned long index)
{
	if (!PageLocked(page))
		BUG();

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        page_cache_get(page);
        spin_lock(&pagecache_lock);
        page->index = index;
        add_page_to_inode_queue(mapping, page);
        add_page_to_hash_queue(page, page_hash(mapping, index));
        lru_cache_add(page);
        spin_unlock(&pagecache_lock);
    }
```

调用参数 mapping 是一个 address_space 结构指针，就是&swapper_space。这种数据结构的定义见 include/linux/fs.h：

==================== include/linux/fs.h 365 375 ====================

```
    struct address_space {
        struct list_head   clean_pages;  /* list of clean pages */
        struct list_head   dirty_pages;  /* list of dirty pages */
        struct list_head   locked_pages; /* list of locked pages */
        unsigned long      nrpages; /* number of total pages */
        struct address_space_operations *a_ops;    /* methods */
        struct inode       *host;        /* owner: inode, block_device */
        struct vm_area_struct   *i_mmap; /* list of private mappings */
        struct vm_area_struct   *i_mmap_shared; /* list of shared mappings */
        spinlock_t         i_shared_lock;  /* and spinlock protecting it */
    };
```

结构中有三个队列头，前两个分别用于“干净”的和“脏”的页面（需要写出），另一个队列头 locked_pages 用于需要暂时锁定在内存不让换出的页面。数据结构 swapper_space 的定义见于 mm/swap_state.c：

==================== mm/swap_state.c 31 37 ====================

```
    struct address_space swapper_space = {
        LIST_HEAD_INIT(swapper_space.clean_pages),
        LIST_HEAD_INIT(swapper_space.dirty_pages),
        LIST_HEAD_INIT(swapper_space.locked_pages),
        0,                  /* nrpages    */
        &swap_aops,
    };
```

结构中的最后一个成分指向另一个数据结构 swap_aops，里面包含了各种 swap 操作的函数指针。

从函数 add_to_page_cache_locked()中可以看到，页面 page 被加入到三个队列中。下面读者会看到，page 结构通过其队列头 list 链入暂存队列 swapper_space，通过指针 next_hash 和双重指针 pprev_hash 链入某个杂凑队列，并通过其队列头 lru 链入 LRU 队列 active_list。

代码中的 page_cache_get()在 include/linux/pagemap.h 中定义为 get_page(page)，实际上只是将页面的使用计数 page->count 加 1。这是在 include/linux/mm.h 中定义的：

==================== include/linux/mm.h 150 150 ====================

```
    #define get_page(p)         atomic_inc(&(p)->count)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/pagemap.h 31 31 ====================

```
#define page_cache_get(x)   get_page(x)
```

先将给定的 page 结构通过 add_page_to_inode_queue()加入到 swapper_space 中的 clean_pages 队列，其代码在 mm/filemap.c 中：

==================== mm/filemap.c 72 79 ====================

```
static inline void add_page_to_inode_queue(struct address_space *mapping, struct page * page)
{
     struct list_head *head = &mapping->clean_pages;

     mapping->nrpages++;
     list_add(&page->list, head);
     page->mapping = mapping;
}
```

可见，链入的是 swapper_space 中的 clean_pages 队列，刚从交换设备读入的页面当然是“干净”页面。为什么这个函数叫 add_page_to_inode_queue 呢？这是因为页面的缓冲不光是为页面交换而设的，文件的读／写也要用到这种缓冲机制。通常来自同一个文件的页面就通过一个 address_space 数据结构来管理，而代表着一个文件的 inode 数据结构中有个成分 i_data，那就是一个 address_space 数据结构。从这个意义上说，用来管理可交换页面的 address_space 数据结构 swapper_space 只是个特例。

然后通过__add_page_to_hash_queue()将其链入到某个杂凑队列中，其代码也在 mm/filemap.c 中：

==================== mm/filemap.c 58 70 ====================

```
static void add_page_to_hash_queue(struct page * page, struct page **p)
{
     struct page *next = *p;

     *p = page;
     page->next_hash = next;
     page->pprev_hash = p;
     if (next)
          next->pprev_hash = &page->next_hash;
     if (page->buffers)
          PAGE_BUG(page);
     atomic_inc(&page_cache_size);
}
```

链入的具体队列取决于杂凑值：

==================== include/linux/pagemap.h 68 68 ====================

```
#define page_hash(mapping,index) (page_hash_table+_page_hashfn(mapping,index))
```

最后将页面的 page 数据结构通过 lru_cache_add()链入到内核中的 LRU 队列 active_list 中，其代码在 mm/swap.c 中：

==================== mm/swap.c 226 241 ====================

```
/**
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * lru_cache_add: add a page to the page lists
 * @page: the page to add
 */
void lru_cache_add(struct page * page)
{
	spin_lock(&pagemap_lru_lock);
	if (!PageLocked(page))
		BUG();
	DEBUG_ADD_PAGE
	add_page_to_active_list(page);
	/* This should be relatively rare */
	if (!page->age)
		deactivate_page_nolock(page);
	spin_unlock(&pagemap_lru_lock);
}
```

这里的 add_page_active_list()是个宏操作，定义于 include/linux/swap.h 内：

==================== include/linux/swap.h 209 215 ====================

```
#define add_page_to_active_list(page) { \
	DEBUG_ADD_PAGE \
	ZERO_PAGE_BUG \
	SetPageActive(page); \
	list_add(&(page)->lru, &active_list); \
	nr_active_pages++; \
}
```

由于 page 数据结构可以通过其同一个队列头结构 lru 链入不同的 LRU 队列，所以需要有 PG_active、PG_inactive_dirty 以及 PG_inactive_clean 等标志位来表明目前是在哪一个队列中。以后读者将看到页面在这些队列间的转移。

存储管理不完全是内核的事，用户进程可以在相当程度上参与对内存的管理，可以在一定的范围内对于其本身的内存管理向内核提出一些要求，例如通过系统调用 mmap()将一文件映射到它的用户空间。特别是特权用户进程，还掌握着对换入／换出机制的全局性控制权，这就是系统调用 swapon()和 swapoff()。调用界面为：

```
swapon (const char *path, int swapflags)
swapoff (const char*path)
```

这两个系统调用是为特权用户进程设置的，用以开始或终止把某个特定的盘区或文件用于页面的换入换出。当所留的盘区和文件都不再用于页面交换时，存储管理的机制就退化到单纯的地址映射和保护。在实践中，这样做有时候是必要的。一些“嵌入式”系统，常常用 Flash Memory（闪存）来代替磁盘介质。对 Flash Memory 的写操作是很麻烦费时的，需要将存储器中的内容先抹去，然后才写入，而抹去的过程又很慢（与磁盘读写相比较）。显然，Flash Memory 是不适合用作页面交换的。所以在这样的系统中应将盘区交换关闭。事实上，在 Linux 内核刚引导进来之初，所有的页面交换都是关闭的，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

内核在初始化期间要执行/etc/rc.d/rc.S 命令文件，而这个文件中的命令行之一就是与系统调用 swapon()相应的实用程序 swapon。只要把这命令行从文件中拿掉就没有页面交换了。

此外，还有几个用于共享内存的系统调用，也是与存储管理有关的。由于习惯上将共享内存归入进程间通讯的范畴，对这几个系统调用将在进程间通讯一章中另行介绍。

2.7 物理页面的分配

上一节中曾经提到，当需要分配若干内存页面时，用于 DMA 的内存页面必须是连续的。其实，为便于管理，特别是出于对物理存储空间“质地”一致性的考虑，即使不是用于 DMA 的内存页面也是连续分配的。

当一个进程需要分配若干连续的物理页面时，可以通过 alloc_pages()来完成。Linux 内核 2.4.0 版的代码中有两个 alloc_pages()，一个在 mm/numa.c 中，另一个在 mm/page_alloc.c 中，编译时根据所定义的条件编译选择项 CONFIG_DISCONTIGMEM 决定取舍。为什么呢？这就是出于前一节中所述对物理存储空间“质地”一致性的考虑。

我们先来看用于 NUMA 结构的 alloc_pages()，其代码在 mm/numa.c 中：

==================== mm/numa.c 43 43 ====================

```
#ifdef CONFIG_DISCONTIGMEM
```

==================== mm/numa.c 91 128 ====================

```
/*
 * This can be refined. Currently, tries to do round robin, instead
 * should do concentratic circle search, starting from current node.
 */
struct page * alloc_pages(int gfp_mask, unsigned long order)
{
	struct page *ret = 0;
	pg_data_t *start, *temp;
#ifndef CONFIG_NUMA
	unsigned long flags;
	static pg_data_t *next = 0;
#endif

	if (order >= MAX_ORDER)
		return NULL;
#ifdef CONFIG_NUMA
	temp = NODE_DATA(numa_node_id());
#else
	spin_lock_irqsave(&node_lock, flags);
	if (!next) next = pgdat_list;
	temp = next;
	next = next->node_next;
	spin_unlock_irqrestore(&node_lock, flags);
#endif
	start = temp;
```

```
    while (temp) {
        if ((ret = alloc_pages_pgdat(temp, gfp_mask, order)))
            return(ret);
        temp = temp->node_next;
    }
    temp = pgdat_list;
    while (temp != start) {
        if ((ret = alloc_pages_pgdat(temp, gfp_mask, order)))
            return(ret);
        temp = temp->node_next;
    }
    return(0);
}
```

首先，对 NUMA 的支持是通过条件编译作为可选项提供的，所以这段代码仅在可选项 CONFIG_DISCONTIGMEM 有定义时才得到编译。不过，这里用来作为条件的是“不连续存储空间”，而不是 CONFIG_NUMA。其实，不连续的物理存储空间是一种广义的 NUMA，因为那说明在最低物理地址和最高物理地址之间存在着空洞，而有空洞的空间当然是非均质的。所以在地址不连续的物理空间也要像在质地不均匀的物理空间那样划分出若干连续（而且均匀）的“节点”。所以，在存储空间不连续的系统中，每个模块都有个若干个节点，因而都有个 pg_data_t 数据结构的队列。

调用时有两个参数。第一个参数 gfp_mask 是个整数，表示采用哪一种分配策略；第二个参数 order 表示所需的物理块大小，可以是 1、2、4、…、直到 2^{MAX_ORDER} 个页面。

在 NUMA 结构的系统中，可以通过宏操作 NUMA_DATA 和 numa_node_id()找到 CPU 所在节点的 pg_data_t 数据结构队列。而在不连续存储空间结构中，则也有个 pg_data_t 数据结构的队列 pgdat_list，分配时轮流从各个节点开始，以求各节点负荷的平衡。

函数中主要的操作在于两个 while 循环，它们分两截（先是从 temp 开始到队列的末尾，然后回头从第一个节点到最初开始的地方）扫描队列中所有的节点，直至在某个节点内分配成功，或彻底失败而返回 0。对于每个节点，调用 alloc_pages_pgdat()试图分配所需的页面，这个函数的代码在 mm/numa.c 中：

==================== mm/numa.c 85 89 ====================

```
static struct page * alloc_pages_pgdat(pg_data_t *pgdat, int gfp_mask,
    unsigned long order)
{
    return __alloc_pages(pgdat->node_zonelists + gfp_mask, order);
}
```

可见，参数 gfp_mask 在这里用作给定节点中数组 node_zonelists[]的下标，决定具体的分配策略。把这段代码与下面用于连续空间 UMA 结构的 alloc_pages()对照一下，就可以看出区别：在连续空间 UMA 结构中只有一个节点 contig_pape_data，而在 NUMA 结构或不连续空间 UMA 结构中则有多个。

连续空间 UMA 结构的 alloc_pages()是在文件 include/linux/mm.h 中定义的：

==================== include/linux/mm.h 343 352 ====================

```
#ifndef CONFIG_DISCONTIGMEM
static inline struct page * alloc_pages(int gfp_mask, unsigned long order)
```

```
{
        /*
         * Gets optimized away by the compiler.
         */
        if (order >= MAX_ORDER)
                return NULL;
        return __alloc_pages(contig_page_data.node_zonelists+(gfp_mask), order);
}
```

与 NUMA 结构的 alloc_papes()相反，这个函数仅在 CONFIG_DISCONTIGMEM 无定义时才得到编译。所以这两个同名的函数只有一个会得到编译。

具体的页面分配由函数__alloc_pages()完成，其代码在 mm/page_alloc.c 中，我们分段阅读：

==================== mm/page_alloc.c 270 315 ====================

[alloc_pages()>__alloc_pages()]

```
/*
 * This is the 'heart' of the zoned buddy allocator:
 */
struct page * __alloc_pages(zonelist_t *zonelist, unsigned long order)
{
        zone_t **zone;
        int direct_reclaim = 0;
        unsigned int gfp_mask = zonelist->gfp_mask;
        struct page * page;

        /*
         * Allocations put pressure on the VM subsystem.
         */
        memory_pressure++;

        /*
         * (If anyone calls gfp from interrupts nonatomically then it
         * will sooner or later tripped up by a schedule().)
         *
         * We are falling back to lower-level zones if allocation
         * in a higher zone fails.
         */

        /*
         * Can we take pages directly from the inactive_clean
         * list?
         */
        if (order == 0 && (gfp_mask & __GFP_WAIT) &&
                        !(current->flags & PF_MEMALLOC))
                direct_reclaim = 1;

        /*
         * If we are about to get low on free pages and we also have
         * an inactive page shortage, wake up kswapd.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 */
	if (inactive_shortage() > inactive_target / 2 && free_shortage())
		wakeup_kswapd(0);
	/*
	 * If we are about to get low on free pages and cleaning
	 * the inactive_dirty pages would fix the situation,
	 * wake up bdflush.
	 */
	else if (free_shortage() && nr_inactive_dirty_pages > free_shortage()
			&& nr_inactive_dirty_pages >= freepages.high)
		wakeup_bdflush(0);

```

调用时有两个参数。第一个参数 zonelist 指向代表着一个具体分配策略的 zonelist_t 数据结构。另一个参数 order 则与前面 alloc_pages()中的相同。全局量 memory_pressure 表示内存页面管理所受的压力，分配内存页面时递增，归还时则递减。这里的局部量 gfp_mask 来自代表着具体分配策略的数据结构，是一些用于控制目的的标志位。如果要求分配的只是单个页面，而且要等待分配完成，又不是用于管理目的，则把一个局部量 direct_reclaim 设成 1，表示可以从相应页面管理区的“不活跃干净页面”缓冲队列中回收。这些页面的内容都已写出至页面交换设备或文件中，只是还保存着页面的内容，使得在需要这个页面的内容时无需再从设备或文件读入，但是当空闲页面短缺时，就顾不得那么多了。由于一般而言这些页面不一定能像真正的空闲页面那样连成块，所以仅在要求分配单个页面时才能从这些页面中回收。此外，当发现可分配页面短缺时，还要唤醒 kswapd 和 bdflush 两个内核线程，让它们设法腾出一些内存页面来（详见“页面的定期换出”）。我们继续往下看：

==================== mm/page_alloc.c 316 340 ====================
[alloc_pages()>__alloc_pages()]

```
try_again:
	/*
	 * First, see if we have any zones with lots of free memory.
	 *
	 * We allocate free memory first because it doesn't contain
	 * any data ... DUH!
	 */
	zone = zonelist->zones;
	for (;;) {
		zone_t *z = *(zone++);
		if (!z)
			break;
		if (!z->size)
			BUG();

		if (z->free_pages >= z->pages_low) {
			page = rmqueue(z, order);
			if (page)
				return page;
		} else if (z->free_pages < z->pages_min &&
				waitqueue_active(&kreclaimd_wait)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    wake_up_interruptible(&kreclaimd_wait);
            }
        }

```

这是对一个分配策略中所规定的所有页面管理区的循环。循环中依次考察各个管理区中空闲页面的总量，如果总量尚在“低水位”以上，就通过 rmqueue()试图从该管理区中分配。要是发现管理区中的空闲页面总量已经降到了最低点，而且有进程（实际上只能是内核线程 kreclaimd）在一个等待队列 kreclaimd_wait 中睡眠，就把它唤醒，让它帮助回收一些页面备用。函数 rmqueue()试图从一个页面管理区分配若干连续的内存页面，其代码在 mm/page_alloc.c 中：

==================== mm/page_alloc.c 172 211 ====================
[alloc_pages()>__alloc_pages()>rmqueue()]

```
static struct page * rmqueue(zone_t *zone, unsigned long order)
{
        free_area_t * area = zone->free_area + order;
        unsigned long curr_order = order;
        struct list_head *head, *curr;
        unsigned long flags;
        struct page *page;

        spin_lock_irqsave(&zone->lock, flags);
        do {
                head = &area->free_list;
                curr = memlist_next(head);

                if (curr != head) {
                        unsigned int index;

                        page = memlist_entry(curr, struct page, list);
                        if (BAD_RANGE(zone,page))
                                BUG();
                        memlist_del(curr);
                        index = (page - mem_map) - zone->offset;
                        MARK_USED(index, curr_order, area);
                        zone->free_pages -= 1 << order;

                        page = expand(zone, page, index, order, curr_order, area);
                        spin_unlock_irqrestore(&zone->lock, flags);

                        set_page_count(page, 1);
                        if (BAD_RANGE(zone,page))
                                BUG();
                        DEBUG_ADD_PAGE
                        return page;
                }
                curr_order++;
                area++;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
207      } while (curr_order < MAX_ORDER);
208      spin_unlock_irqrestore(&zone->lock, flags);
209
210      return NULL;
211  }
```

以前讲过，代表物理页面的 page 数据结构，以双向链的形式链接在管理区的某个空闲队列中。分配页面时当然要把它从队列中摘链，而摘链的过程是不容许其它的进程、其它的处理器（如果有的话）来打扰的。所以要用 spin_lock_irqsave()将相应的分区加上锁，不容许打扰。管理区结构中的空闲区 zone->free_area 是个结构数组，所以 zone->free_area + oder 就指向链接所需大小的物理内存块的队列头。主要的操作是在一个 do-while 循环中进行。它首先在恰好满足大小要求的队列里分配，如果不行的话就试试更大的（指物理内存块）队列中分配，成功的话，就把分配到的大块中剩余的部分分解成小块而链入相应的队列（通过 196 行的 expand()）。

第 188 行中的 memlist_entry()从一个非空的队列里取第一个结构 page 元素，然后通过 memlist_del()将其从队列中摘除。对此，我们已在第 1 章中作过解释。

函数 expand()是在同一文件（mm/page_alloc.c）中定义的：

==================== mm/page_alloc.c 150 169 ====================

[alloc_pages()>__alloc_pages()>rmqueue()>expand()]

```
150  static inline struct page * expand (zone_t *zone, struct page *page,
151       unsigned long index, int low, int high, free_area_t * area)
152  {
153      unsigned long size = 1 << high;
154
155      while (high > low) {
156          if (BAD_RANGE(zone,page))
157              BUG();
158          area--;
159          high--;
160          size >>= 1;
161          memlist_add_head(&(page)->list, &(area)->free_list);
162          MARK_USED(index, high, area);
163          index += size;
164          page += size;
165      }
166      if (BAD_RANGE(zone,page))
167          BUG();
168      return page;
169  }
```

调用参数表中的 low 对应于表示所需物理块大小的 order，而 high 则对应于表示当时空闲区队列（也就是从中得到能满足要求的物理块的队列）的 curr_order。当两者相符时，从 155 行开始的 while 循环就被跳过了。若是分配到的物理块大于所需的大小（不可能小于所需的大小），那就将该物理块链入低一档也就是物理块大小减半的空闲块队列中去，并相应设置该空闲区队列的位图，这是在第 158 行至 162 行中完成的。然后从该物理块中切去一半，而以其后半部作为一个新的物理块（第 163 和 164 行），

而后开始下一轮循环也就是处理更低一档的空闲块队列。这样，最后必有 high 与 low 两者相等，也就是实际剩下的物理块与要求恰好相符的时候，循环就结束了。

就这样，rmqueue()一直往上扫描，直到成功或者最终失败。如果 rmqueue()失败，则__alloc_pages()通过其 for 循环降格以求，接着试分配策略中规定的下一个管理区，直到成功，或者碰到了空指针而最终失败（见 327 行）。如果分配成功了，则__alloc_pages()返回一个 page 结构指针，指向页面块中第一个页面的 page 结构，并且该 page 结构中的使用计数 count 为 1。如果每次分配的都是单个的页面（order 为 0），则自然每个页面的使用计数都是 1。

要是给定分配策略中所有的页面管理区都失败了，那就只好“加大力度”再试，一是降低对页面管理区中保持“水位”的要求，二是把缓冲在管理区中的“不活跃干净页面”也考虑进去。我们再往下看__alloc_pages()的代码（mm/page_alloc.c）。

==================== mm/page_alloc.c 341 364 ====================

[alloc_pages()>__alloc_pages()]

```
	/*
	 * Try to allocate a page from a zone with a HIGH
	 * amount of free + inactive_clean pages.
	 *
	 * If there is a lot of activity, inactive_target
	 * will be high and we'll have a good chance of
	 * finding a page using the HIGH limit.
	 */
	page = __alloc_pages_limit(zonelist, order, PAGES_HIGH, direct_reclaim);
	if (page)
		return page;

	/*
	 * Then try to allocate a page from a zone with more
	 * than zone->pages_low free + inactive_clean pages.
	 *
	 * When the working set is very large and VM activity
	 * is low, we're most likely to have our allocation
	 * succeed here.
	 */
	page = __alloc_pages_limit(zonelist, order, PAGES_LOW, direct_reclaim);
	if (page)
		return page;

```

这里先以参数 PAGES_HIGH 调用__alloc_pages_limit()；如果还不行就再加大力度，改以 PAGES_LOW 再调用一次。函数__alloc_pages_limit()的代码也在 mm/page_alloc.c 中：

==================== mm/page_alloc.c 213 267 ====================

[alloc_pages()>__alloc_pages()>__alloc_pages_limit()]

```
#define PAGES_MIN	0
#define PAGES_LOW	1
#define PAGES_HIGH	2
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

/*
 * This function does the dirty work for __alloc_pages
 * and is separated out to keep the code size smaller.
 * (suggested by Davem at 1:30 AM, typed by Rik at 6 AM)
 */
static struct page * __alloc_pages_limit(zonelist_t *zonelist,
                        unsigned long order, int limit, int direct_reclaim)
{
        zone_t **zone = zonelist->zones;

        for (;;) {
                zone_t *z = *(zone++);
                unsigned long water_mark;

                if (!z)
                        break;
                if (!z->size)
                        BUG();

                /*
                 * We allocate if the number of free + inactive_clean
                 * pages is above the watermark.
                 */
                switch (limit) {
                        default:
                        case PAGES_MIN:
                                water_mark = z->pages_min;
                                break;
                        case PAGES_LOW:
                                water_mark = z->pages_low;
                                break;
                        case PAGES_HIGH:
                                water_mark = z->pages_high;
                }

                if (z->free_pages + z->inactive_clean_pages > water_mark) {
                        struct page *page = NULL;
                        /* If possible, reclaim a page directly. */
                        if (direct_reclaim && z->free_pages < z->pages_min + 8)
                                page = reclaim_page(z);
                        /* If that fails, fall back to rmqueue. */
                        if (!page)
                                page = rmqueue(z, order);
                        if (page)
                                return page;
                }
        }

```

```
    /* Found nothing. */
    return NULL;
}
```

这个函数的代码与前面__alloc_pages()中的 for 循环在逻辑下只是稍有不同，我们把它留给读者。其中 reclaim_page()从页面管理区的 inactive_clean_list 队列中回收页面，其代码在 mm/vmscan.c 中，我们把它列出在“页面的定期换出”一节的末尾，读者可以在学习了页面的换入和换出以后自己阅读。注意调用这个函数的条件是参数 direct_reclaim 非 0，所以要求分配的一定是单个页面。

还是不行的话，那就说明这些管理区中的页面已经严重短缺了，让我们看看__alloc_pages()是如何对付的：

==================== mm/page_alloc.c 365 399 ====================

[alloc_pages()>__alloc_pages()]

```
    /*
     * OK, none of the zones on our zonelist has lots
     * of pages free.
     *
     * We wake up kswapd, in the hope that kswapd will
     * resolve this situation before memory gets tight.
     *
     * We also yield the CPU, because that:
     * - gives kswapd a chance to do something
     * - slows down allocations, in particular the
     *   allocations from the fast allocator that's
     *   causing the problems ...
     * - ... which minimises the impact the "bad guys"
     *   have on the rest of the system
     * - if we don't have __GFP_IO set, kswapd may be
     *   able to free some memory we can't free ourselves
     */
    wakeup_kswapd(0);
    if (gfp_mask & __GFP_WAIT) {
         __set_current_state(TASK_RUNNING);
         current->policy |= SCHED_YIELD;
         schedule();
    }

    /*
     * After waking up kswapd, we try to allocate a page
     * from any zone which isn't critical yet.
     *
     * Kswapd should, in most situations, bring the situation
     * back to normal in no time.
     */
    page = __alloc_pages_limit(zonelist, order, PAGES_MIN, direct_reclaim);
    if (page)
         return page;

```

首先是唤醒内核线程 kswapd，让它设法换出一些页面。如果分配策略表明对于要求分配的页面是志在必得，分配不到时宁可等待，就让系统来一次调度，并且让当前进程为其它进程让一下路。这样，一来让 kswapd 有可能立即被调度运行，二来其它进程也有可能会释放出一些页面，再说也可减缓了要求分配页面的速度，减轻了压力。当请求分配页面的进程再次被调度运行时，或者分配策略表明不允许等待时，就以参数 PAGES_MIN 再调用一次__alloc_pages_limit()。可是，要是再失败呢？这时候就要看是谁在要求分配内存页面了。如果要求分配页面的进程（或线程）是 kswapd 或 kreclaimd，本身就是“内存分配工作者”，要求分配内存页面的目的是执行公务，是要更好地分配内存页面，这当然比一般的进程更重要。这些进程的 task_struct 结构中 flags 字段的 PF_MEMALLOC 标志位为 1。我们先看对于一般进程，即 PF_MEMALLOC 标志位为 0 的进程的对策。

==================== mm/page_alloc.c 400 477 ====================

[alloc_pages()>__alloc_pages()]

```
	/*
	 * Damn, we didn't succeed.
	 *
	 * This can be due to 2 reasons:
	 * - we're doing a higher-order allocation
	 *   --> move pages to the free list until we succeed
	 * - we're /really/ tight on memory
	 *   --> wait on the kswapd waitqueue until memory is freed
	 */
	if (!(current->flags & PF_MEMALLOC)) {
		/*
		 * Are we dealing with a higher order allocation?
		 *
		 * Move pages from the inactive_clean to the free list
		 * in the hope of creating a large, physically contiguous
		 * piece of free memory.
		 */
		if (order > 0 && (gfp_mask & __GFP_WAIT)) {
			zone = zonelist->zones;
			/* First, clean some dirty pages. */
			current->flags |= PF_MEMALLOC;
			page_launder(gfp_mask, 1);
			current->flags &= ~PF_MEMALLOC;
			for (;;) {
				zone_t *z = *(zone++);
				if (!z)
					break;
				if (!z->size)
					continue;
				while (z->inactive_clean_pages) {
					struct page * page;
					/* Move one page to the free list. */
					page = reclaim_page(z);
					if (!page)
						break;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
                __free_page(page);
                /* Try if the allocation succeeds. */
                page = rmqueue(z, order);
                if (page)
                    return page;
            }
        }
    }
    /*
     * When we arrive here, we are really tight on memory.
     *
     * We wake up kswapd and sleep until kswapd wakes us
     * up again. After that we loop back to the start.
     *
     * We have to do this because something else might eat
     * the memory kswapd frees for us and we need to be
     * reliable. Note that we don't loop back for higher
     * order allocations since it is possible that kswapd
     * simply cannot free a large enough contiguous area
     * of memory *ever*.
     */
    if ((gfp_mask & (__GFP_WAIT|__GFP_IO)) == (__GFP_WAIT|__GFP_IO)) {
        wakeup_kswapd(1);
        memory_pressure++;
        if (!order)
            goto try_again;
    /*
     * If __GFP_IO isn't set, we can't wait on kswapd because
     * kswapd just might need some IO locks /we/ are holding ...
     *
     * SUBTLE: The scheduling point above makes sure that
     * kswapd does get the chance to free memory we can't
     * free ourselves...
     */
    } else if (gfp_mask & __GFP_WAIT) {
        try_to_free_pages(gfp_mask);
        memory_pressure++;
        if (!order)
            goto try_again;
    }

}

```

分配内存页面失败的原因可能是两方面的，一种可能是可分配页面的总量实在已经太少了；另一种是总量其实还不少，但是所要求的页面块大小却不能满足，此时往往有不少单个的页面在管理区的 inactive_clean_pages 队列中，如果加以回收就有可能拼装起较大的页面块。同时，可能还有些“脏 ”页面在全局的 inactive_dirty_pages 队列中，把脏页面的内容写出到交换设备上或文件中，就可以使它

们变成“干净”页面而加以回收。所以，针对第二种可能，代码中通过 page_launder()把“脏页”面“洗净”（详见“页面的定期换出”），然后通过一个 for 循环在各个页面管理区中回收和释放“干净”页面。具体的回收和释放是通过一个 while 循环完成的。在通过__free_page()释放页面时会把空闲页面拼装起尽可能大的页面块，所以在每回收了一个页面以后都要调用 rmqueue()试一下，看看是否已经能满足要求。值得注意的是，这里在调用 page_launder()期间把当前进程的 PF_MEMALLOC 标志位设成 1，使其有了“执行公务”时的特权。为什么要这样做呢？这是因为在 page_launder()中也会要求分配一些临时性的工作页面，不把 PF_MEMALLOC 标志位设成 1 就可能递归地进入这里的 409～476 行。

如果回收了这样的页面以后还是不行，那就是可分配页面的总量不够了。这时候一种办法是唤醒 kswapd，而要求分配页面的进程则睡眠等待，由 kswapd 在完成了一轮运行之后再反过来唤醒要求分配页面的进程。然后，如果要求分配的是单个页面，就通过 goto 语句转回__alloc_pages()开头处的标号 try_again 处。另一种办法是直接调用 try_to_free_pages()，这个函数本来是 kswapd 调用的。

那么，如果是“执行公务”呢？或者，虽然不是执行公务，但已想尽了一切办法，采取了一切措施，只不过因为要求分配的是成块的页面才没有转回前面的标号 try_again 处。

前面我们看到，一次次加大力度调用__alloc_pages_limit()时，实际上还是有所保留的。例如，最后一次以 PAGES_MIN 为参数，此时判断是否可以分配的准则是管理区中可分配页面的“水位”高于 z->pages_min。之所以还留着一点“老本”，是为应付紧急状况，而现在已到了“不惜血本”的时候了。我们继续往下看__alloc_pages()的代码。

==================== mm/page_alloc.c 478 521 ====================

[alloc_pages()>__alloc_pages()]

```
478      /*
479       * Final phase: allocate anything we can!
480       *
481       * Higher order allocations, GFP_ATOMIC allocations and
482       * recursive allocations (PF_MEMALLOC) end up here.
483       *
484       * Only recursive allocations can use the very last pages
485       * in the system, otherwise it would be just too easy to
486       * deadlock the system...
487       */
488      zone = zonelist->zones;
489      for (;;) {
490           zone_t *z = *(zone++);
491           struct page * page = NULL;
492           if (!z)
493                break;
494           if (!z->size)
495                BUG();
496
497           /*
498            * SUBTLE: direct_reclaim is only possible if the task
499            * becomes PF_MEMALLOC while looping above. This will
500            * happen when the OOM killer selects this task for
501            * instant execution...
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                 */
                if (direct_reclaim) {
                        page = reclaim_page(z);
                        if (page)
                                return page;
                }

                /* XXX: is pages_min/4 a good amount to reserve for this? */
                if (z->free_pages < z->pages_min / 4 &&
                                !(current->flags & PF_MEMALLOC))
                        continue;
                page = rmqueue(z, order);
                if (page)
                        return page;
        }

        /* No luck.. */
        printk(KERN_ERR "__alloc_pages: %lu-order allocation failed.\n", order);
        return NULL;
}
```

如果连这也失败，那一定是系统有问题了。

读者也许会说：好家伙，分配一个（或几个）内存页面有这么麻烦，那 CPU 还有多少时间能用于实质性的计算呢？要知道我们这里是假定分配页面的努力“屡战屡败”，而又“屡败屡战”，这才有这么多艰苦卓绝的努力。实际上，绝大多数的分配页面操作都是在分配策略所规定的第一个页面管理区中就成功了。不过，从这里我们可以看到设计一个系统需要何等周密的考虑。

2.8　页面的定期换出

这个情景比较长，读者得有点耐心。

为了避免总是在 CPU 忙碌的时候，也就是在缺页异常发生的时候，临时再来搜寻可供换出的内存页面并加以换出，Linux 内核定期地检查并且预先将若干页面换出，腾出空间，以减轻系统在缺页异常发生时的负担。当然，由于无法确切地预测页面的使用，即使这样做了也还是不能完全杜绝在缺页异常发生时内存没有空闲页面，而只好临时寻找可换出页面的可能。但是，这样毕竟可以减少其发生的概率。并且，通过选择适当的参数，例如每隔多久换出一次，每次换出多少页面，可以使得在缺页异常发生时必须临时寻找页面换出的情况实际上很少发生。为此，在 Linux 内核中设置了一个专司定期将页面换出的“守护神 ” kswapd。

从原理上说，kswapd 相当于一个进程，有其自身的进程控制块 task_struct 结构，跟其它进程一样受内核的调度。而正因为内核将它按进程来调度，就可以让它在系统相对空闲的时候来运行。不过，与普通的进程相比，kswapd 还是有其特殊性。首先，它没有自己独立的地址空间，所以在近代操作系统理论中称为“线程”（thread）以示区别。那么，kswapd 使用谁的地址空间呢？它使用的是内核的空间。在这一点上，它与中断服务程序相似。其次，它的代码是静态地连接在内核中的，可以直接调用

内核中的各种子程序，而不像普通的进程那样只能通过系统调用，使用预先定义好的一组功能。

本节讲述 kswapd 受内核调度而运行并走完一条例行路线的全过程。

线程 kswapd 的源代码基本上都在 mm/vmscan.c 中。先来看它的建立：

==================== mm/vmscan.c 1146 1153 ====================

```
1146 static int __init kswapd_init(void)
1147 {
1148       printk("Starting kswapd v1.8\n");
1149       swap_setup();
1150       kernel_thread(kswapd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
1151       kernel_thread(kreclaimd, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
1152       return 0;
1153 }
```

函数 kswapd_init()是在系统初始化期间受到调用的，它主要做两件事。第一件是在 swap_setup()中根据物理内存的大小设定一个全局量 page_cluster：

==================== mm/swap.c 293 305 ====================

[kswapd_init()>swap_setup()]

```
293  /*
294   * Perform any setup for the swap system
295   */
296  void __init swap_setup(void)
297  {
298       /* Use a smaller cluster for memory <16MB or <32MB */
299       if (num_physpages < ((16 * 1024 * 1024) >> PAGE_SHIFT))
300            page_cluster = 2;
301       else if (num_physpages < ((32 * 1024 * 1024) >> PAGE_SHIFT))
302            page_cluster = 3;
303       else
304            page_cluster = 4;
305  }
```

这是一个跟磁盘设备驱动有关的参数。由于读磁盘时先要经过寻道，并且寻道是个比较费时间的操作，所以如果每次只读一个页面是不经济的。比较好的办法是既然读了就干脆多读几个页面，称为“预读”。但是预读意味着每次需要暂存更多的内存页面，所以需要决定一个适当的数量，而根据物理内存本身的大小来确定这个参数显然是合理的。第二件事就是创建线程 kswapd，这是由 kernel_thread()完成的。这里还创建了另一个线程 kreclaimd，也是跟存储管理有关，不过不像 kswapd 那么复杂和重要，所以我们暂且把它放在一边。关于建立线程的详情请参阅进程管理一章，这里暂且假定线程 kswapd 就此建立了，并且从函数 kswapd()开始执行。其代码在 mm/vmscan.c 中：

==================== mm/vmscan.c 947 1046 ====================

```
947  /*
948   * The background pageout daemon, started as a kernel thread
949   * from the init process.
950   *
951   * This basically trickles out pages so that we have _some_
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * free memory available even if there is no other activity
 * that frees anything up. This is needed for things like routing
 * etc, where we otherwise might have all activity going on in
 * asynchronous contexts that cannot page things out.
 *
 * If there are applications that are active memory-allocators
 * (most normal use), this basically shouldn't matter.
 */
int kswapd(void *unused)
{
     struct task_struct *tsk = current;

     tsk->session = 1;
     tsk->pgrp = 1;
     strcpy(tsk->comm, "kswapd");
     sigfillset(&tsk->blocked);
     kswapd_task = tsk;

     /*
      * Tell the memory management that we're a "memory allocator",
      * and that if we need more memory we should get access to it
      * regardless (see "__alloc_pages()"). "kswapd" should
      * never get caught in the normal page freeing logic.
      *
      * (Kswapd normally doesn't need memory anyway, but sometimes
      * you need a small amount of memory in order to be able to
      * page out something else, and this flag essentially protects
      * us from recursively trying to free more memory as we're
      * trying to free the first piece of memory in the first place).
      */
     tsk->flags |= PF_MEMALLOC;

     /*
      * Kswapd main loop.
      */
     for (;;) {
          static int recalc = 0;

          /* If needed, try to free some memory. */
          if (inactive_shortage() || free_shortage()) {
               int wait = 0;
               /* Do we need to do some synchronous flushing? */
               if (waitqueue_active(&kswapd_done))
                    wait = 1;
               do_try_to_free_pages(GFP_KSWAPD, wait);
          }

          /*
           * Do some (very minimal) background scanning. This
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
             * will scan all pages on the active list once
             * every minute. This clears old referenced bits
             * and moves unused pages to the inactive list.
             */
            refill_inactive_scan(6, 0);

            /* Once a second, recalculate some VM stats. */
            if (time_after(jiffies, recalc + HZ)) {
                 recalc = jiffies;
                 recalculate_vm_stats();
            }

            /*
             * Wake up everybody waiting for free memory
             * and unplug the disk queue.
             */
            wake_up_all(&kswapd_done);
            run_task_queue(&tq_disk);

            /*
             * We go to sleep if either the free page shortage
             * or the inactive page shortage is gone. We do this
             * because:
             * 1) we need no more free pages   or
             * 2) the inactive pages need to be flushed to disk,
             *    it wouldn't help to eat CPU time now ...
             *
             * We go to sleep for one second, but if it's needed
             * we'll be woken up earlier...
             */
            if (!free_shortage() || !inactive_shortage()) {
                 interruptible_sleep_on_timeout(&kswapd_wait, HZ);
            /*
             * If we couldn't free enough memory, we see if it was
             * due to the system just not having enough memory.
             * If that is the case, the only solution is to kill
             * a process (the alternative is enternal deadlock).
             *
             * If there still is enough memory around, we just loop
             * and try free some more memory...
             */
            } else if (out_of_memory()) {
                 oom_kill();
            }
       }
}
```

在一些简单的初始化操作以后，程序便进入一个无限循环。在每次循环的末尾一般都会调用 interruptible_sleep_on_timeout()进入睡眠，让内核自由地调度别的进程运行。但是内核在一定时间以后

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

又会唤醒并调度 kswapd 继续运行，这时候 kswapd 就又回到这无限循环开始的地方。那么，这“一定时间”是多长呢，这就是常数 HZ。HZ 决定了内核中每秒钟有多少次时钟中断。用户可以在编译内核前的系统配置阶段改变其数值，但是一经编译就定下来了。所以，在调用 interruptible_sleep_on_timeout()时的参数为 HZ，表示 1 秒钟以后又要调度 kswapd 继续运行。换言之，对 interruptible_sleep_on_timeout()的调用一进去就得 1 秒钟以后才回来。但是，在有些情况下内核也会在不到 1 秒钟时就把它唤醒，那样 kswapd 就会提前返回而开始新的一轮循环。所以，这个循环至少每隔 1 秒钟执行一遍，这就是 kswapd 的例行路线。

那么，kswapd 在这至少每秒一次的例行路线中做些什么呢？可以把它分成两部分。第一部分是在发现物理页面已经短缺的情况下才进行的，目的在于预先找出若干页面，且将这些页面的映射断开，使这些物理页面从活跃状态转入不活跃状态，为页面的换出作好准备。第二部分是每次都要执行的，目的在于把已经处于不活跃状态的“脏”页面写入交换设备，使它们成为不活跃“干净”页面继续缓冲，或进一步回收一些这样的页面成为空闲页面。

先看第一部分，首先检查内存中可供分配或周转的物理页面是否短缺：

==================== mm/vmscan.c 805 822 ====================

[kswapd()>inactive_shortage()]

```
/*
 * How many inactive pages are we short?
 */
int inactive_shortage(void)
{
	int shortage = 0;

	shortage += freepages.high;
	shortage += inactive_target;
	shortage -= nr_free_pages();
	shortage -= nr_inactive_clean_pages();
	shortage -= nr_inactive_dirty_pages;

	if (shortage > 0)
		return shortage;

	return 0;
}
```

系统中应该维持的物理页面供应量由两个全局量确定，那就是 freepages.high 和 inactive_target，分别为空闲页面的数量和不活跃页面的数量，二者之和为正常情况下潜在的供应量。而这些内存页面的来源则有三个方面。一方面是当前尚存的空闲页面，这是立即就可以分配的页面。这些页面分散在各个页面管理区中，并且合并成地址连续、大小为 2、4、8、…、2^N 个页面的页面块，其数量由 nr_free_pages()加以统计。另一方面是现有的不活跃“干净”页面，这些页面本质上也是马上就可以分配的页面，但是页面中的内容可能还会用到，所以多保留一些这样的页面有助于减少从交换设备的读入。这些页面也分散在各个页面管理区中，但并不合并成块，其数量由 nr_inactive_clean_pages()加以统计。最后是现有的不活跃“脏”页面，这些页面要先加以“净化”，即写入交换设备以后才能投入分配。这种页面

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

全都在同一个队列中，内核中的全局量 nr_inactive_dirty_pages 记录着当前此类页面的数量。上述两个函数的代码都在 mm/page_alloc.c 中，也都比较简单，读者可以自己阅读。

不过，光维持潜在的物理页面供应总量还不够，还要通过 free_shortage()检查是否有某个具体管理区中有严重的短缺，即直接可供分配的页面数量（除不活跃“脏”页面以外）是否小于一个最低限度。这个函数的代码在 mm/vmscan.c 中，我们也把它留给读者。

如果发现可供分配的内存页面短缺，那就要设法释放和换出若干页面，这是通过 do_try_to_free_pages()完成的。不过在此之前还要调用 waitqueue_active()，看看 kswapd_done 队列中是否有函数在等待执行，并把查看的结果作为参数传递给 do_try_to_free_pages()。在第 3 章中，读者将看到内核中有几个特殊的队列，内核中各个部分（主要是设备驱动）可以把一些低层函数挂入这样的队列，使得这些函数在某种事件发生时就能得到执行。而 kswapd_done，就正是这样的一个队列。凡是挂入这个队列的函数，在 kswapd 每完成一趟例行的操作时就能得到执行。这里的 inline 函数 waitqueue_active()就是查看是否有函数在这个队列中等待执行。其定义在 include/linux/wait.h 中：

==================== include/linux/wait.h 152 161 ====================
[kswapd()>waitqueue_active()]

```
static inline int waitqueue_active(wait_queue_head_t *q)
{
#if WAITQUEUE_DEBUG
     if (!q)
          WQ_BUG();
     CHECK_MAGIC_WQHEAD(q);
#endif

     return !list_empty(&q->task_list);
}
```

下面就是调用 do_try_to_free_pages()，试图腾出一些内存页面。其代码在 mm/vmscan.c 中：

==================== mm/vmscan.c 907 941 ====================
[kswapd()>do_try_to_free_pages()]

```
static int do_try_to_free_pages(unsigned int gfp_mask, int user)
{
     int ret = 0;

     /*
      * If we're low on free pages, move pages from the
      * inactive_dirty list to the inactive_clean list.
      *
      * Usually bdflush will have pre-cleaned the pages
      * before we get around to moving them to the other
      * list, so this is a relatively cheap operation.
      */
     if (free_shortage() || nr_inactive_dirty_pages > nr_free_pages() +
               nr_inactive_clean_pages())
          ret += page_launder(gfp_mask, user);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /*
         * If needed, we move pages from the active list
         * to the inactive list. We also "eat" pages from
         * the inode and dentry cache whenever we do this.
         */
        if (free_shortage() || inactive_shortage()) {
             shrink_dcache_memory(6, gfp_mask);
             shrink_icache_memory(6, gfp_mask);
             ret += refill_inactive(gfp_mask, user);
        } else {
             /*
              * Reclaim unused slab cache memory.
              */
             kmem_cache_reap(gfp_mask);
             ret = 1;
        }

        return ret;
}
```

将活跃页面的映射断开，使之转入不活跃状态，甚至进而换出到交换设备上，是不得已而为之，因为谁也不能精确地预测到底哪一些页面是合适的换出对象。虽然一般而言“最近最少用到”是个有效的准则，但也并不是“放诸四海而皆准”。所以，能够不动“现役”页面是最理想的。基于这样的考虑，这里所作的是先易后难，逐步加强力度。首先是调用 page_launder()，试图把已经转入不活跃状态的“脏”页面“洗净”，使它们变成立即可以分配的页面。函数名中的“ launder”，就是“洗衣工”的意思。这个函数一方面（基本上）定期地受到 kswapd()的调用，一方面在每当需要分配内存页面，而又无页面可供分配时，临时地受到调用。其代码在 mm/vmscan.c 中：

==================== mm/vmscan.c 465 670 ====================

[kswapd()>do_try_to_free_pages()>page_launder()]

```
/**
 * page_launder - clean dirty inactive pages, move to inactive_clean list
 * @gfp_mask: what operations we are allowed to do
 * @sync: should we wait synchronously for the cleaning of pages
 *
 * When this function is called, we are most likely low on free +
 * inactive_clean pages. Since we want to refill those pages as
 * soon as possible, we'll make two loops over the inactive list,
 * one to move the already cleaned pages to the inactive_clean lists
 * and one to (often asynchronously) clean the dirty inactive pages.
 *
 * In situations where kswapd cannot keep up, user processes will
 * end up calling this function. Since the user process needs to
 * have a page before it can continue with its allocation, we'll
 * do synchronous page flushing in that case.
 *
 * This code is heavily inspired by the FreeBSD source code. Thanks
 * go out to Matthew Dillon.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 */
#define MAX_LAUNDER         (4 * (1 << page_cluster))
int page_launder(int gfp_mask, int sync)
{
     int launder_loop, maxscan, cleaned_pages, maxlaunder;
     int can_get_io_locks;
     struct list_head * page_lru;
     struct page * page;

     /*
      * We can only grab the IO locks (eg. for flushing dirty
      * buffers to disk) if __GFP_IO is set.
      */
     can_get_io_locks = gfp_mask & __GFP_IO;

     launder_loop = 0;
     maxlaunder = 0;
     cleaned_pages = 0;

dirty_page_rescan:
     spin_lock(&pagemap_lru_lock);
     maxscan = nr_inactive_dirty_pages;
     while ((page_lru = inactive_dirty_list.prev) != &inactive_dirty_list &&
                   maxscan-- > 0) {
          page = list_entry(page_lru, struct page, lru);

          /* Wrong page on list?! (list corruption, should not happen) */
          if (!PageInactiveDirty(page)) {
               printk("VM: page_launder, wrong page on list.\n");
               list_del(page_lru);
               nr_inactive_dirty_pages--;
               page->zone->inactive_dirty_pages--;
               continue;
          }

          /* Page is or was in use?  Move it to the active list. */
          if (PageTestandClearReferenced(page) || page->age > 0 ||
                   (!page->buffers && page_count(page) > 1) ||
                   page_ramdisk(page)) {
               del_page_from_inactive_dirty_list(page);
               add_page_to_active_list(page);
               continue;
          }

          /*
           * The page is locked. IO in progress?
           * Move it to the back of the list.
           */
          if (TryLockPage(page)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        list_del(page_lru);
                        list_add(page_lru, &inactive_dirty_list);
                        continue;
                }

                /*
                 * Dirty swap-cache page? Write it out if
                 * last copy..
                 */
                if (PageDirty(page)) {
                        int (*writepage)(struct page *) = page->mapping->a_ops->writepage;
                        int result;

                        if (!writepage)
                                goto page_active;

                        /* First time through? Move it to the back of the list */
                        if (!launder_loop) {
                                list_del(page_lru);
                                list_add(page_lru, &inactive_dirty_list);
                                UnlockPage(page);
                                continue;
                        }

                        /* OK, do a physical asynchronous write to swap.  */
                        ClearPageDirty(page);
                        page_cache_get(page);
                        spin_unlock(&pagemap_lru_lock);

                        result = writepage(page);
                        page_cache_release(page);

                        /* And re-start the thing.. */
                        spin_lock(&pagemap_lru_lock);
                        if (result != 1)
                                continue;
                        /* writepage refused to do anything */
                        set_page_dirty(page);
                        goto page_active;
                }

                /*
                 * If the page has buffers, try to free the buffer mappings
                 * associated with this page. If we succeed we either free
                 * the page (in case it was a buffercache only page) or we
                 * move the page to the inactive_clean list.
                 *
                 * On the first round, we should free all previously cleaned
                 * buffer pages
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         */
        if (page->buffers) {
            int wait, clearedbuf;
            int freed_page = 0;
            /*
             * Since we might be doing disk IO, we have to
             * drop the spinlock and take an extra reference
             * on the page so it doesn't go away from under us.
             */
            del_page_from_inactive_dirty_list(page);
            page_cache_get(page);
            spin_unlock(&pagemap_lru_lock);

            /* Will we do (asynchronous) IO? */
            if (launder_loop && maxlaunder == 0 && sync)
                wait = 2;/* Synchrounous IO */
            else if (launder_loop && maxlaunder-- > 0)
                wait = 1;/* Async IO */
            else
                wait = 0;/* No IO */

            /* Try to free the page buffers. */
            clearedbuf = try_to_free_buffers(page, wait);

            /*
             * Re-take the spinlock. Note that we cannot
             * unlock the page yet since we're still
             * accessing the page_struct here...
             */
            spin_lock(&pagemap_lru_lock);

            /* The buffers were not freed. */
            if (!clearedbuf) {
                add_page_to_inactive_dirty_list(page);

            /* The page was only in the buffer cache. */
            } else if (!page->mapping) {
                atomic_dec(&buffermem_pages);
                freed_page = 1;
                cleaned_pages++;

            /* The page has more users besides the cache and us. */
            } else if (page_count(page) > 2) {
                add_page_to_active_list(page);

            /* OK, we "created" a freeable page. */
            } else /* page->mapping && page_count(page) == 2 */ {
                add_page_to_inactive_clean_list(page);
                cleaned_pages++;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
630             }
631
632             /*
633              * Unlock the page and drop the extra reference.
634              * We can only do it here because we ar accessing
635              * the page struct above.
636              */
637             UnlockPage(page);
638             page_cache_release(page);
639
640             /*
641              * If we're freeing buffer cache pages, stop when
642              * we've got enough free memory.
643              */
644             if (freed_page && !free_shortage())
645                 break;
646             continue;
647         } else if (page->mapping && !PageDirty(page)) {
648             /*
649              * If a page had an extra reference in
650              * deactivate_page(), we will find it here.
651              * Now the page is really freeable, so we
652              * move it to the inactive_clean list.
653              */
654             del_page_from_inactive_dirty_list(page);
655             add_page_to_inactive_clean_list(page);
656             UnlockPage(page);
657             cleaned_pages++;
658         } else {
659 page_active:
660             /*
661              * OK, we don't know what to do with the page.
662              * It's no use keeping it here, so we move it to
663              * the active list.
664              */
665             del_page_from_inactive_dirty_list(page);
666             add_page_to_active_list(page);
667             UnlockPage(page);
668         }
669     }
670     spin_unlock(&pagemap_lru_lock);
```

代码中的局部量 cleaned_papes 用来累计被“洗清”的页面数量。另一个局部量 launder_loop 用来控制扫描不活跃“脏”页面队列的次数。在第一趟扫描时 launder_loop 为 0，如果有必要进行第二趟扫描，则将其设成 1 并转回到标号 dirty_page_rescan 处（502 行），开始又一次扫描。

对不活跃“脏”页面队列的扫描是通过一个 while 循环（505 行）进行的。由于在循环中会把有些页面从当前位置移到队列的尾部，所以除沿着链接指针扫描外还要对数量加以控制，才能避免重复处理同一页面，甚至陷入死循环，这就是变量 maxscan 的作用。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

对于队列中的每一个页面，首先要检查它的 PG_inactive_dirty 标志位为 1，否则就根本不应该出现在这个队列中；这一定是出了什么毛病，所以把它从队列中删除（见 512 行）。除此之外，对于正常的不活跃“脏”页面，则要依次作下述的检查并作相应的处理。

(1) 有些页面虽然已经进入不活跃“脏”页面队列，但是由于情况已经变化，或者当初进入这个队列本来就是“冤假错案”，因而需要回到活跃页面队列中（519～525 行）。这样的页面有：
页面在进入了不活跃“脏”页面队列之后又受到了访问，即发生了以此页面为目标的缺页异常，从而恢复了该页面的映射。
页面的“寿命”还未耗尽。页面的 page 结构中有个字段 age，其数值与页面受访问的频繁程度有关。后面我们还要回到这个话题。
页面并不用作读／写文件的缓冲，而页面的使用计数却又大于 1。这说明页面在至少一个进程的映射表中有映射。如前所述，一个页面的使用计数在分配时设成 1，以后对该页面的每一次使用都使这个计数加 1，包括将页面用作读／写文件的缓冲。如果一个页面没有用作读／写文件的缓冲，那么只要计数大于 1 就必定还有进程在使用这个页面。
页面在受到进程用户空间映射的同时又用于 ramdisk，即用内存空间来模拟磁盘，这种页面当然不应该换出到磁盘上。

(2) 页面已被锁住（531 行），所以 TryLockPage()返回 1，这表明正在对此页面进行操作，如输入／输出，这样的页面应该留在不活跃“脏”页面队列中，但是把它移到队列的尾部。注意，对于未被锁住的页面，现在已经锁上了。

(3) 如果页面仍是“脏”的（541 行），即 page 结构中的 PG_dirty 标志位为 1，则原则上将其写出到交换设备上，但还有些特殊情况要考虑（541～571 行）。首先，所属的 address_space 数据结构必须提供页面写出操作的函数，否则就只好转到 page_active 处，将页面送回活跃页面队列中。对于一般的页面交换，所属的 address_space 数据结构为 swapper_space，其 address_space_operations 结构为 swap_aops，所提供的页面写出操作为 swap_writepage()，过这一“关”是没有问题的。在第一趟扫描中，只是把页面移到同一队列的尾部，而并不写出页面（531～535 行）。如果进行第二趟扫描的话，那就真的要把页面写出去了。写之前先通过 ClearPageDirty()把页面的 PG_dirty 标志位清成 0，然后通过由所属 address_space 数据结构所提供的函数把页面写出去。根据页面的不同使用目的，例如普通的用户空间页面，或者通过 mmap()建立的文件映射以及文件系统的读／写缓冲，具体的操作也不一样。这个写操作可能是同步的（当前进程睡眠，等待写出完成），也可能是异步的，但总是需要一定的时间才能完成，在此期间内核有可能再次进入 page_launder()，所以需要防止把这个页面再写出一次。这就是把页面的 PG_dirty 标志位清成 0 的目的。这样，就不会把同一个页面写出两次了（见 541 行）。此外，还要考虑页面写出失败的可能，具体的函数在写出失败时应该返回 1，使 page_launder()可以恢复页面的 PG_dirty 标志位并将其退还给活跃页面队列中（569～570 行）。顺便提一下，这里在调用具体的 writepage 函数时先通过 page_cache_get()递增页面的使用计数，从这个函数返回后再通过 page_cache_release()递减这个计数，表示在把页面写出的期间多了一个“用户”。注意这里并没有立即把写出的页面转移到不活跃“干净”页面队列中，而只是把它的 PG_dirty 标志位清成了 0。还要注意，如果 CPU 到达了代码中的 582 行，则页面的 PG_dirty 标志位必定是 0，这个页面一定是在以前的扫描中写出而变“干净”的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

(4) 如果页面不再是“脏”的，并且又是用作文件读 / 写缓冲的页面（582～647 行），则先使它脱离不活跃“脏”页面队列，再通过 try_to_free_buffers()试图将页面释放。如果不能释放则根据返回值将其退回不活跃“脏”页面队列，或者链入活跃页面队列，或者不活跃“干净”页面队列。如果释放成功，则页面的使用计数已经在 try_to_free_buffers()中减 1，638 行的 page_cache_release()再使其减 1 就达到了 0，从而最终将页面释放回到空闲页面队列中。如果成功地释放了一个页面，并已发现系统中的空闲页面已经不再短缺，那么扫描就可以结束了（见 644 和 645 行）。否则继续扫描。函数 try_to_free_buffers()的代码在 fs/buffer.c 中，读者可以在学习了“文件系统”一章以后自行阅读。

(5) 如果页面不再是“脏”的，并且在某个 address_space 数据结构的队列中，这就是已经“洗清 ”了的页面，所以把它转移到所属区间的不活跃“干净”页面队列中。

(6) 最后，如果不属于上述的任何一种情况（658 行），那就是无法处理的页面，所以把它退回活跃页面队列中。

完成了一趟扫描以后，还要根据系统中空闲页面是否短缺、以及调用参数的 gfp_mask 中的 __GFP_IO 标志位是否为 1，来决定是否进行第二趟扫描。

==================== mm/vmscan.c 671 697 ====================

[kswapd()>do_try_to_free_pages()>page_launder()]

```
671
672 	/*
673 	 * If we don't have enough free pages, we loop back once
674 	 * to queue the dirty pages for writeout. When we were called
675 	 * by a user process (that /needs/ a free page) and we didn't
676 	 * free anything yet, we wait synchronously on the writeout of
677 	 * MAX_SYNC_LAUNDER pages.
678 	 *
679 	 * We also wake up bdflush, since bdflush should, under most
680 	 * loads, flush out the dirty pages before we have to wait on
681 	 * IO.
682 	 */
683 	if (can_get_io_locks && !launder_loop && free_shortage()) {
684 		launder_loop = 1;
685 		/* If we cleaned pages, never do synchronous IO. */
686 		if (cleaned_pages)
687 			sync = 0;
688 		/* We only do a few "out of order" flushes. */
689 		maxlaunder = MAX_LAUNDER;
690 		/* Kflushd takes care of the rest. */
691 		wakeup_bdflush(0);
692 		goto dirty_page_rescan;
693 	}
694
695 	/* Return the number of pages moved to the inactive_clean list. */
696 	return cleaned_pages;
697 }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

如果决定进行第二趟扫描，就转回到 502 行标号 dirty_page_rescan 处。注意这里把 launder_loop 设成了 1，以后就不可能再回过去又扫描一次了。所以每次调用 page_launder()最多是作两趟扫描。

回到 do_try_to_free_pages()的代码中，经过 page_launder()以后，如果可分配的物理页面数量仍然不足，那就要进一步设法回收页面了。不过，也并不是单纯地从各个进程的用户空间所映射的物理页面中回收，而是从四个方面回收，这就是这里所调用三个函数（shrink_dcache_memory()、shrink_icache_memory()、refill_inactive()），以及等一下将会看到的 kmem_cache_reap()的意图。在“文件系统”一章中，读者将会看到，在打开文件的过程中要分配和使用代表着目录项的 dentry 数据结构，还有代表着文件索引节点的 inode 数据结构。这些数据结构在文件关闭以后并不立即释放，而是放在 LRU 队列中作为后备，以防在不久将来的文件操作中又要用到。这样，经过一段时间以后，就有可能积累起大量的 dentry 数据结构和 inode 数据结构，占用数量可观的物理页面。这时，就要通过 shrink_dcache_memory()和 shrink_icache_memory()适当加以回收，以维持这些数据结构与物理页面间的“生态平衡”。另一方面，除此以外，内核在运行中也需要动态地分配使用很多数据结构，内核中对此采用了一种称为“ slab”的管理机制。以后读者会看到，这种机制就好像是向存储管理“批发”物理页面，然后切割成小块“零售”。随着系统的运行，对这种物理页面的实际需求也在动态地变化。但是 slab 管理机制也是倾向于分配和保持更多的空闲物理页面，而不热衷于退还这些页面，所以过一段时间就要通过 kmem_cache_reap()来“收割”。读者可以在学习了“文件系统”后回过来自己阅读前两个函数的代码，我们在这里则集中关注 refill_inactive()，其代码在 mm/vmscan.c 中：

==================== mm/vmscan.c 824 905 ====================
[kswapd()>do_try_to_free_pages()>refill_inactive()]

```
/*
 * We need to make the locks finer granularity, but right
 * now we need this so that we can do page allocations
 * without holding the kernel lock etc.
 *
 * We want to try to free "count" pages, and we want to
 * cluster them so that we get good swap-out behaviour.
 *
 * OTOH, if we're a user process (and not kswapd), we
 * really care about latency. In that case we don't try
 * to free too many pages.
 */
static int refill_inactive(unsigned int gfp_mask, int user)
{
     int priority, count, start_count, made_progress;

     count = inactive_shortage() + free_shortage();
     if (user)
          count = (1 << page_cluster);
     start_count = count;

     /* Always trim SLAB caches when memory gets low. */
     kmem_cache_reap(gfp_mask);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	priority = 6;
	do {
		made_progress = 0;

		if (current->need_resched) {
			__set_current_state(TASK_RUNNING);
			schedule();
		}

		while (refill_inactive_scan(priority, 1)) {
			made_progress = 1;
			if (--count <= 0)
				goto done;
		}

		/*
		 * don't be too light against the d/i cache since
		 * refill_inactive() almost never fail when there's
		 * really plenty of memory free.
		 */
		shrink_dcache_memory(priority, gfp_mask);
		shrink_icache_memory(priority, gfp_mask);

		/*
		 * Then, try to page stuff out..
		 */
		while (swap_out(priority, gfp_mask)) {
			made_progress = 1;
			if (--count <= 0)
				goto done;
		}

		/*
		 * If we either have enough free memory, or if
		 * page_launder() will be able to make enough
		 * free memory, then stop.
		 */
		if (!inactive_shortage() || !free_shortage())
			goto done;

		/*
		 * Only switch to a lower "priority" if we
		 * didn't make any useful progress in the
		 * last loop.
		 */
		if (!made_progress)
			priority--;
	} while (priority >= 0);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    /* Always end on a refill_inactive.., may sleep... */
    while (refill_inactive_scan(0, 1)) {
        if (--count <= 0)
            goto done;
    }

done:
    return (count < start_count);
}
```

参数 user 是从 kswapd 传下来的，表示是否有函数在 kswapd_done 队列中等待执行，这个因素决定回收物理页面的过程是否可以慢慢来，所以对本次要回收的页面数量有影响。

首先通过 kmem_cache_reap()“收割”由 slab 机制管理的空闲物理页面，相对而言这是动作最小的，读者可以在学习了“内核工作缓冲区的管理”一节以后自己阅读这个函数的代码。

然后，就是一个 do-while 循环。循环从优先级最低的 6 级开始，逐步加大“力度”直到 0 级，结果或者达到了目标，回收的数量够了；或者在最高优先级时还是达不到目标，那也只好算了（到缺页中断真的发生时情况也许有了改变）。

在循环中，每次开头都要检查一下当前进程的 task_struct 结构中的 need_resched 是否为 1。如果是，就说明某个中断服务程序要求调度，所以调用 schedule()让内核进行一次调度，但是在此之前把本进程的状态设置成 TASK_RUNNING，表达要继续运行的意愿。读者在第 4 章中将会看到，task_struct 结构中的 need_resched 是为强制调度而设置的，每当 CPU 结束了一次系统调用或中断服务、从系统空间返回用户空间时就会检查这个标志。可是，kswapd 是个内核线程，永远不会“返回用户空间”，这样就有可能绕过这个机制而占住 CPU 不放，所以只能靠它“自律”，自己在可能需要较长时间的操作之前检查这个标志并调用 schedule()。

那么，在循环中做些什么呢？主要是两件事。一件是通过 refill_inactive_scan()扫描活跃页面队列。试图从中找到可以转入不活跃状态的页面；另一件是通过 swap_out()找出一个进程，然后扫描其映射表，从中找出可以转入不活跃状态的页面。此外，还要再试试用于 dentry 结构和 inode 结构的页面。先看 refill_inactive_scan()的代码，这个函数在 mm/vmscan.c 中：

==================== mm/vmscan.c 699 769 ====================

```
/**
 * refill_inactive_scan - scan the active list and find pages to deactivate
 * @priority: the priority at which to scan
 * @oneshot: exit after deactivating one page
 *
 * This function will scan a portion of the active list to find
 * unused pages, those pages will then be moved to the inactive list.
 */
int refill_inactive_scan(unsigned int priority, int oneshot)
{
    struct list_head * page_lru;
    struct page * page;
    int maxscan, page_active = 0;
    int ret = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    /* Take the lock while messing with the list... */
    spin_lock(&pagemap_lru_lock);
    maxscan = nr_active_pages >> priority;
    while (maxscan-- > 0 && (page_lru = active_list.prev) != &active_list) {
         page = list_entry(page_lru, struct page, lru);

         /* Wrong page on list?! (list corruption, should not happen) */
         if (!PageActive(page)) {
              printk("VM: refill_inactive, wrong page on list.\n");
              list_del(page_lru);
              nr_active_pages--;
              continue;
         }

         /* Do aging on the pages. */
         if (PageTestandClearReferenced(page)) {
              age_page_up_nolock(page);
              page_active = 1;
         } else {
              age_page_down_ageonly(page);
              /*
               * Since we don't hold a reference on the page
               * ourselves, we have to do our test a bit more
               * strict then deactivate_page(). This is needed
               * since otherwise the system could hang shuffling
               * unfreeable pages from the active list to the
               * inactive_dirty list and back again...
               *
               * SUBTLE: we can have buffer pages with count 1.
               */
              if (page->age == 0 && page_count(page) <=
                             (page->buffers ? 2 : 1)) {
                   deactivate_page_nolock(page);
                   page_active = 0;
              } else {
                   page_active = 1;
              }
         }
         /*
          * If the page is still on the active list, move it
          * to the other end of the list. Otherwise it was
          * deactivated by age_page_down and we exit successfully.
          */
         if (page_active || PageActive(page)) {
              list_del(page_lru);
              list_add(page_lru, &active_list);
         } else {
              ret = 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
762                 if (oneshot)
763                     break;
764             }
765         }
766         spin_unlock(&pagemap_lru_lock);
767
768         return ret;
769 }
```

就像对“脏”页面队列的扫描一样，这里也通过一个局部量 maxscan 来控制扫描的页面数量。不过这里扫描的不一定是整个活跃页面队列，而是根据调用参数 priority 的值扫描其中一部分，只有在 priority 为 0 时才扫描整个队列（见 716 行）。对于所扫描的页面，首先也要验证确实属于活跃页面（见 721 行）。然后，根据页面是否受到了访问（见 729 行），决定增加或减少页面的寿命。如果减少页面寿命以后到达了 0，那就说明这个页面已经很长时间没有受到访问，因而已经耗尽了寿命。不过，光是耗尽了寿命还不足以把页面从活跃状态转入不活跃状态，还得看是否还有用户空间映射。如果页面并不用作文件系统的读 / 写缓冲，那么只要页面的使用计数大于 1 就说明还有用户空间映射，还不能转入不活跃状态（见 744 行），这样的页面在通过 swap_out()扫描相应进程的映射表时才能转入不活跃状态。对于还不能转入不活跃状态的页面，要将其从队列中的当前位置移到队列的尾部。反之，如果成功地将一个页面转入了不活跃状态，则根据参数 oneshot 的值决定是否继续扫描。一般来说，在活跃页面队列中的页面使用计数都大于 1。而当 swap_out()断开一个页面的映射而使其转入不活跃状态时，则已经将页面转入不活跃页面队列，因而不在这个队列中了。可是，就如代码中的注释所言，确实存在着特殊的情况，在“页面的换入”中就可以看到。

再看 swap_out()，那是在 mm/vmscan.c 中在义的：

==================== mm/vmscan.c 297 378 ====================

[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()]

```
297 /*
298  * Select the task with maximal swap_cnt and try to swap out a page.
299  * N.B. This function returns only 0 or 1.  Return values != 1 from
300  * the lower level routines result in continued processing.
301  */
302 #define SWAP_SHIFT 5
303 #define SWAP_MIN 8
304
305 static int swap_out(unsigned int priority, int gfp_mask)
306 {
307     int counter;
308     int __ret = 0;
309
310     /*
311      * We make one or two passes through the task list, indexed by
312      * assign = {0, 1}:
313      *   Pass 1: select the swappable task with maximal RSS that has
314      *        not yet been swapped out.
315      *   Pass 2: re-assign rss swap_cnt values, then select as above.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 *
	 * With this approach, there's no need to remember the last task
	 * swapped out.  If the swap-out fails, we clear swap_cnt so the
	 * task won't be selected again until all others have been tried.
	 *
	 * Think of swap_cnt as a "shadow rss" - it tells us which process
	 * we want to page out (always try largest first).
	 */
	counter = (nr_threads << SWAP_SHIFT) >> priority;
	if (counter < 1)
		counter = 1;

	for (; counter >= 0; counter--) {
		struct list_head *p;
		unsigned long max_cnt = 0;
		struct mm_struct *best = NULL;
		int assign = 0;
		int found_task = 0;
	select:
		spin_lock(&mmlist_lock);
		p = init_mm.mmlist.next;
		for (; p != &init_mm.mmlist; p = p->next) {
			struct mm_struct *mm = list_entry(p, struct mm_struct, mmlist);
			if (mm->rss <= 0)
				continue;
			found_task++;
			/* Refresh swap_cnt? */
			if (assign == 1) {
				mm->swap_cnt = (mm->rss >> SWAP_SHIFT);
				if (mm->swap_cnt < SWAP_MIN)
					mm->swap_cnt = SWAP_MIN;
			}
			if (mm->swap_cnt > max_cnt) {
				max_cnt = mm->swap_cnt;
				best = mm;
			}
		}

		/* Make sure it doesn't disappear */
		if (best)
			atomic_inc(&best->mm_users);
		spin_unlock(&mmlist_lock);

		/*
		 * We have dropped the tasklist_lock, but we
		 * know that "mm" still exists: we are running
		 * with the big kernel lock, and exit_mm()
		 * cannot race with us.
		 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (!best) {
            if (!assign && found_task > 0) {
                assign = 1;
                goto select;
            }
            break;
        } else {
            __ret = swap_out_mm(best, gfp_mask);
            mmput(best);
            break;
        }
    }
    return __ret;
}
```

这个函数的主体是一个 for 循环，循环的次数取决于 counter，而 counter 又是根据内核中进程（包括线程）的个数和调用 swap_out()时优先级（最初为 6 级，逐次上升至 0 级）计算而得的。当优先级为 0 时，counter 就等于（nr_threads << SWAP_SHIFT），即 32 × nr_threads，这里 nr_threads 为当前系统中进程的数量。这个数值决定了把页面换出去的“决心”有多大，即代码中外层循环的次数。参数 gfp_mask 中是一些控制信息。

在每次循环中，程序试图从所有的进程中找到一个最合适的进程 best。找到了就扫描这个进程的页面映射表，将符合条件的页面暂时断开对内存页面的映射，或进一步将页面转入不活跃状态，为把这些页面换出到交换设备上作好准备。

这里还应指出，这个函数虽然叫“ swap_out”，但实际上只是为把一些页面换出到交换设备上作好准备，而并不一定是物理意义上的页面换出，所以在下面的叙述中所谓“换出”是广义的。那么，根据什么准则来找“最合适”的进程呢？可以说是“劫富济贫”与“轮流坐庄”相结合。每个进程都有其自身的虚存空间，空间中已经分配并建立了映射的页面构成一个集合。但是在任何一个给定的时刻，该集合中的每一个页面所对应的物理页面不一定都在内存中，在内存中的往往只是一个子集。这个子集称为“驻内页面集合”（resident set），其大小称为 rss。在存储管理结构 mm_struct 中有一个成分就是 rss。以前我们在讲到这个结构时把 rss 跳过了，因为说来话长。而现在到了结合情景和源代码加以说明的时候了。

代码中的内层 for 循环表示从第二个进程开始扫描所有的进程。内核中所有的 task_struct 结构都以双向链连接成一个队列。而进程 init_task 是内核中的第一个进程，是所有其它进程的祖宗。只要内核还在运行，这个进程就“永远不落”。所以，从 init_task.next_task 始至 init_task 止，就是扫描除第一个进程外的所有进程。扫描的目的是从中找出 mm->swap_cnt 为最大的进程。每个 mm_struct 结构中的这个数值，是在把所有进程的页面资源时都处理了一遍，从而每个 mm_struct 结构中的这个数值都变成了 0 的时候设置好了的，反映了当时该进程占用内存页面的数量 mm->rss。这就好像一次“人口普查”。随后，每次考察和处理了这个进程的一个页面，就将其 mm->swap_cnt 减 1，直至最后变成 0。所以，mm->rss 反映了一个进程占用的内存页面数量，而 mm->wap_cnt 反映了该进程在一轮换出内行页面的努力中尚未受到考察的页面数量。只要在这一轮中至少还有一个进程的页面尚未受到考察，就一定能找到一个“最佳对象”。一直到所有进程的 mm->swap_cnt 都变成了 0，从而扫描下来竟找不到一个“ best ”

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

时（439～444 行），再把这里的局部量 assign 置成 1，再扫描一遍。这一次将每个进程当前的 mm->rss 拷贝到 mm->swap_cnt 中，然后再从最富有的进程开始。但是，所谓尚未受到考察的页面数量并不包括最近一次“人口普查”以后因页面异常而换入（或恢复映射）的页面，这些页面的数量要到下一次“人口普查”以后才会反映出来。就每个进程的角度而言，对内存页面的占用存在着两个方向上的运动：一个方向是因页面异常而有更多的页面建立起或恢复起映射；另一个方面则是周期性地受到 swap_out() 的考察而被切断若干页面的映射。这两个运动的结合决定了一个进程在特定时间内对内存页面的占用。

找到一个“最佳对象 ” best 以后，就要依次考察该进程的映射表，将符合条件的页面换出去。

页面的换出具体是由 swap_out_mm()来完成的。当 swap_out_mm()成功地换出一个页面时返回 1，否则返回 0，返回负数则为异常。在操作之前先通过 356 行的 atomic_inc()递增 mm_struct 结构中的使用计数 mm_users，待完成以后再由 373 行的 mmput()将其还原，使这个数据结构在操作的期间多了一个用户，从而不会在中途被释放。

函数 swap_out_mm()的代码也在 mm/vmscan.c 中：

==================== mm/vmscan.c 257 295 ====================

[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()]

```
static int swap_out_mm(struct mm_struct * mm, int gfp_mask)
{
     int result = 0;
     unsigned long address;
     struct vm_area_struct* vma;

     /*
      * Go through process' page directory.
      */

     /*
      * Find the proper vm-area after freezing the vma chain
      * and ptes.
      */
     spin_lock(&mm->page_table_lock);
     address = mm->swap_address;
     vma = find_vma(mm, address);
     if (vma) {
          if (address < vma->vm_start)
               address = vma->vm_start;

          for (;;) {
               result = swap_out_vma(mm, vma, address, gfp_mask);
               if (result)
                    goto out_unlock;
               vma = vma->vm_next;
               if (!vma)
                    break;
               address = vma->vm_start;
          }
     }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
288       /* Reset to 0 when we reach the end of address space */
289       mm->swap_address = 0;
290       mm->swap_cnt = 0;
291
292  out_unlock:
293       spin_unlock(&mm->page_table_lock);
294       return result;
295  }
```

首先，mm->swap_address 表示在执行的过程中要接着考察的页面地址。最初时该地址为 0，到所有的页面都已考察了一遍的时候就又清成 0（见 289 行）。程序在一个 for 循环中根据当前的这个地址找到其所在的虚存区域 vma，然后就调用 swap_out_vma()试图换出一个页面。如果成功（返回 1），这一次任务就完成了。否则就试下一个虚存区间。就这样一层一层地往下调用，经过 swap_out_vma()、swap_out_pgd()、swap_out_pmd()，一直到 try_to_swap_out()，试图换出由一个页面表项 pte 所指向的内存页面。中间这几个函数都在同一个文件中，读者可以自行阅读。这里我们直接来看 try_to_swap_out()，因为这是关键所在。下面，我们一步一步来看它的各个片断：

==================== mm/vmscan.c 27 56 ====================
[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()]

```
/*
 * The swap-out functions return 1 if they successfully
 * threw something out, and we got a free page. It returns
 * zero if it couldn't do anything, and any other value
 * indicates it decreased rss, but the page was shared.
 *
 * NOTE! If it sleeps, it *must* return 1 to make sure we
 * don't continue with the swap-out. Otherwise we may be
 * using a process that no longer actually exists (it might
 * have died while we slept).
 */
static int try_to_swap_out(struct mm_struct * mm, struct vm_area_struct* vma, unsigned long address, pte_t * page_table, int gfp_mask)
{
     pte_t pte;
     swp_entry_t entry;
     struct page * page;
     int onlist;

     pte = *page_table;
     if (!pte_present(pte))
          goto out_failed;
     page = pte_page(pte);
     if ((!VALID_PAGE(page)) || PageReserved(page))
          goto out_failed;

     if (!mm->swap_cnt)
          return 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
54
55        mm->swap_cnt--;
56
```

首先要说明，参数 page_table 实际上指向一个页面表项、而不是页面表，参数名 page_table 有些误导。把这个表项的内容赋给变量 pte 以后，就通过 pte_present()来测试该表项所指的物理页面是否在内存中，如果不在内存中就转向 out_failed，本次操作就失败了：

==================== mm/vmscan.c 106 107 ====================

```
106  out_failed:
107          return 0;
```

当 try_to_swap_out()返回 0 时，其上一层的程序就会跳过这个页面，而试着换出同一个页面表中映射的下一个页面。如果一个页面表已经穷尽，就再往上退一层试下一个页面表。

反之，如果物理页面确在内存中，就通过 pte_pages()将页面表项的内容换算成指向该物理内存页面的 pagp 结构的指针。由于所有的 page 结构都在 mem_map 数组中，所以(page - mem_map)就是该页面的序号（数组中的下标）。要是这个序号大于最大的物理内存页面序号 max_mapnr，那就不是一个有效的物理页面，这种情况通常是因为物理页面在外部设备（例如网络接口卡）上，所以也跳过这一项。

==================== include/asm-i386/page.h 118 118 ====================

```
118  #define VALID_PAGE(page)    ((page - mem_map) < max_mapnr)
```

此外，对于保留在内存中不允许换出的物理页面也要跳过。

跳过了这两种特殊情况，就要具体地考察一个页面了，所以将 mm->swap_cnt 减 1。继续往下看 try_to_swap_out()的代码：

==================== mm/vmscan.c 57 74 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()]

```
57        onlist = PageActive(page);
58        /* Don't look at this pte if it's been accessed recently. */
59        if (ptep_test_and_clear_young(page_table)) {
60             age_page_up(page);
61             goto out_failed;
62        }
63        if (!onlist)
64             /* The page is still mapped, so it can't be freeable... */
65             age_page_down_ageonly(page);
66
67        /*
68         * If the page is in active use by us, or if the page
69         * is in active use by others, don't unmap it or
70         * (worse) start unneeded IO.
71         */
72        if (page->age > 0)
73             goto out_failed;
74
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

内存页面的 page 结构中，字段 flags 中的各种标志位反映着页面的当前状态，其中的 PG_active 标志位表示当前这个页面是否“活跃”，即是否仍在 active_list 队列中：

==================== include/linux/mm.h 230 230 ====================

```
#define PageActive(page)    test_bit(PG_active, &(page)->flags)
```

一个可交换的物理页面一定在某个 LRU 队列中，不在 active_list 队列中就说明一定在 inactive_dirty_list 中或某个 inactive_clean_list 中，等一下就要使用测试的结果。

一个映射中的物理页面是否应该换出，取决于这个页面最近是否受到了访问。这是通过 inline 函数 ptep_test_and_clear_young()测试（并清 0）的，其定义在 include/asm-i386/pgtable.h 中：

==================== include/asm-i386/pgtable.h 285 285 ====================

```
static inline  int ptep_test_and_clear_young(pte_t *ptep)    { return
test_and_clear_bit(_PAGE_BIT_ACCESSED, ptep); }
```

如前所述，页面表项中有个_PAGE_ACCESSED 标志位。当 i386 CPU 的内存映射机制在通过一个页面表项将一个虚存地址映射成一个物理地址，进而访问这个物理地址时，就会自动将该表项的_PAGE_ACCESSED 标志位设成 1。所以，如果 pte_young()返回 1，就表示从上一次对同一个页面表项调用 try_to_swap_out()至今，该页面至少已经被访问过一次，所以说页面还“年轻”。一般而言，最近受到过访问就预示着在不久的将来也会受到访问，所以不宜将其换出。取得了此项信息以后，就将页面表项中的_PAGE_ACCESSED 标志位清成 0，再把它写回页面表项，为下一次再来测试这个标志位作好准备。

如果页面还“年轻”，那就肯定不是要加以换出的对象，所以也要转到 out_failed。不过，在转到 out_failed 上前还要做一点事情：如果页面还活跃，就要通过 SetPageReferenced()将 page 数据结构中的 PG_referenced 标志位置成 1。也就是说，将页面表项中表示受到过访问的信息转移至页面的数据结构中。而要是页面不在活跃页面队列中，则通过明 age_page_up()增加页面可以留下来“以观后效”的时间，因为毕竟这个页面最近已受到过访问。

==================== mm/swap.c 125 138 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()>age_page_up()]

```
void age_page_up(struct page * page)
{
	/*
	 * We're dealing with an inactive page, move the page
	 * to the active list.
	 */
	if (!page->age)
		activate_page(page);

	/* The actual page aging bit */
	page->age += PAGE_AGE_ADV;
	if (page->age > PAGE_AGE_MAX)
		page->age = PAGE_AGE_MAX;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

转到 out_failed 以后，就在那里返回 0，让更高层的程序跳过这个页面。这样，到下一轮又轮到这个进程和这个页面时，如果同一页面表项 pte 中的_PAGE_ACCESSED 标志位仍然为 0，那就表示不再“年轻”了。读者也许会问，既然这个页面是有映射的（否则不会出现在目标进程的映射表中并且在内存中），怎么又会不在活跃页面队列中呢？以后读者就会在 do_swap_page()中看到，当因页面异常而恢复一个不活跃页间的映射时，并不立即把它转入活跃页面队列，而把这项工作留给前面看到的 page_launder()，让其在系统比较空闲时再来处理，所以这样的页面有可能不在活跃队列中。

如果页面已不“年轻”，那就要进一步考察了。当然，也不能因为这个页面在过去一个周期中未受到访问就马上把它换出去，还要给它一个“留职察看”的机会。察看多久呢？那就是 pape->age 的值，即页面的寿命。如果页面不在活跃队列中则还要先通过 age_page_down_ageonly()减少其寿命（mm/swap.c）：

==================== mm/swap.c 103 110 ====================

```
/*
 * We use this (minimal) function in the case where we
 * know we can't deactivate the page (yet).
 */
void age_page_down_ageonly(struct page * page)
{
	page->age /= 2;
}
```

只要 page->age 尚未达到 0，就还不能将此页面换出，所以也要转到 out_failed。

经过上面这些筛选，这个页面原则上已经是可以换出的对象了。我们继续往下看代码：

==================== mm/vmscan.c 75 108 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()]

```
	if (TryLockPage(page))
		goto out_failed;

	/* From this point on, the odds are that we're going to
	 * nuke this pte, so read and clear the pte.  This hook
	 * is needed on CPUs which update the accessed and dirty
	 * bits in hardware.
	 */
	pte = ptep_get_and_clear(page_table);
	flush_tlb_page(vma, address);

	/*
	 * Is the page already in the swap cache? If so, then
	 * we can just drop our reference to it without doing
	 * any IO - it's already up-to-date on disk.
	 *
	 * Return 0, as we didn't actually free any real
	 * memory, and we should just continue our scan.
	 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
94        if (PageSwapCache(page)) {
95              entry.val = page->index;
96              if (pte_dirty(pte))
97                    set_page_dirty(page);
98  set_swap_pte:
99              swap_duplicate(entry);
100             set_pte(page_table, swp_entry_to_pte(entry));
101 drop_pte:
102             UnlockPage(page);
103             mm->rss--;
104             deactivate_page(page);
105             page_cache_release(page);
106 out_failed:
107             return 0;
108       }
```

下面对 page 数据结构的操作涉及需要互斥，或者说独占的条件下进行的操作，所以这里通过 TryLockPage()将 page 数据锁住（include/linux/mm.h）：

==================== include/linux/mm.h 183 183 ====================

```
183  #define TryLockPage(page)   test_and_set_bit(PG_locked, &(page)->flags)
```

如果返回值为 1，即表示 PG_locked 标志位原来就已经是 1，已经被别的进程先锁住了，此时就不能继续处理这个 page 数据结构，而又只好失败返回。

加锁成功以后，就可以根据页面的不同情况作换出的准备了。

首先通过 ptep_get_and_clear()再读一次页面表项的内容，并把表项的内容清成 0，暂时撤销该页面的映射。前面在 45 行已经读了一次页面表项的内容，为什么现在还要再读一次，而不仅仅是把表项清 0 呢？在多处理器系统中，目标进程有可能正在另一个 CPU 上运行，所以其映射表项的内容有可能已经改变。

如果页面的 page 数据结构已经在为页面换入／换出而设置的队列中，即数据结构 swapper_space 内的队列中，那么页面的内容已经在交换设备上，只要把映射暂时断开，表示目标进程已经同意释放这个页面，就可以了。不过，为页面换入／换出而设置的队列也分为“干净”和“脏”两个，所以如果页面已经受过写访问就要通过 set_page_dirty()将其转入“脏”页面队列。宏操作 PageSwapCache()的定义为（include/linux/mm.h）：

==================== include/linux/mm.h 217 217 ====================

```
217  #define PageSwapCache(page) test_bit(PG_swap_cache, &(page)->flags)
```

标志位 PG_swap_cache 为 1 表示 page 结构在 swapper_space 队列中，也说明相应的页面是个普通的换入／换出页面。此时 page 结构中的 index 字段是一个 32 位的索引项 swp_entry_t，实际上是指向页面在交换设备上的映象的指引。函数 swap_duplicate()的作用，一者是要对索引项的内容作一些检验，二者是要递增相应盘上页面的共享计数，其代码在 mm/swapfile.c 中：

==================== mm/swapfile.c 820 871 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()>swap_duplicate()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * Verify that a swap entry is valid and increment its swap map count.
 * Kernel_lock is held, which guarantees existance of swap device.
 *
 * Note: if swap_map[] reaches SWAP_MAP_MAX the entries are treated as
 * "permanent", but will be reclaimed by the next swapoff.
 */
int swap_duplicate(swp_entry_t entry)
{
        struct swap_info_struct * p;
        unsigned long offset, type;
        int result = 0;

        /* Swap entry 0 is illegal */
        if (!entry.val)
                goto out;
        type = SWP_TYPE(entry);
        if (type >= nr_swapfiles)
                goto bad_file;
        p = type + swap_info;
        offset = SWP_OFFSET(entry);
        if (offset >= p->max)
                goto bad_offset;
        if (!p->swap_map[offset])
                goto bad_unused;
        /*
         * Entry is valid, so increment the map count.
         */
        swap_device_lock(p);
        if (p->swap_map[offset] < SWAP_MAP_MAX)
                p->swap_map[offset]++;
        else {
                static int overflow = 0;
                if (overflow++ < 5)
                        printk("VM: swap entry overflow\n");
                p->swap_map[offset] = SWAP_MAP_MAX;
        }
        swap_device_unlock(p);
        result = 1;
out:
        return result;

bad_file:
        printk("Bad swap file entry %08lx\n", entry.val);
        goto out;
bad_offset:
        printk("Bad swap offset entry %08lx\n", entry.val);
        goto out;
bad_unused:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        printk("Unused swap offset entry in swap_dup %08lx\n", entry.val);
        goto out;
}
```

以前讲过，数据结构 swp_entry_t 实际上是 32 位无符号整数，其内容不可能全是 0，但是最低位却一定是 0，最高的（24 位）位段 offset 为设备上的页面序号，其余的（7 位）位段 type 则其实是交换设备本身的序号。以前还讲过，其中的位段 type 实际上与“类型”毫无关系，而是代表着交换设备的序号。以此为下标，就可在内核中的数组 swap_info 中找到相应交换设备的 swap_info_struct 数据结构。这个数据结构中的数组 swap_map[]，则记录着交换设备上各个页面的共享计数。由于正在处理中的页面原来就已经在交换设备上，其计数显然不应为 0，否则就错了；另一方面，递增以后也不应达到 SWAP_MAP_MAX。递增盘上页面的共享计数表示这个页面现在多了一个用户。

回到 try_to_swap_out()的代码中，100 行调用 set_pte()，把这个指向盘上页面的索引项置入相应的页面表项，原先对内存页面的映射就变成了对盘上页面的映射。这样，当执行到标号 drop_pte 的地方，目标进程的驻内页面集合 rss 中就减少了一个页面。由于我们这个物理页面断开了一个映射，很可能已经满足了变成不活跃页面的条件，所以在调用 deactivate_page()时有条件地将其设置成不活跃状态，并将页面的 page 结构从活跃页面队列转移到某个不活跃页面队列（mm/swap.c）：

==================== mm/swap.c 189 194 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()>deactivate_page()]

```
void deactivate_page(struct page * page)
{
        spin_lock(&pagemap_lru_lock);
        deactivate_page_nolock(page);
        spin_unlock(&pagemap_lru_lock);
}
```

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()>deactivate_page()>deactivate_page_nolock()]

```
/**
 * (de)activate_page - move pages from/to active and inactive lists
 * @page: the page we want to move
 * @nolock - are we already holding the pagemap_lru_lock?
 *
 * Deactivate_page will move an active page to the right
 * inactive list, while activate_page will move a page back
 * from one of the inactive lists to the active list. If
 * called on a page which is not on any of the lists, the
 * page is left alone.
 */
void deactivate_page_nolock(struct page * page)
{
        /*
         * One for the cache, one for the extra reference the
         * caller has and (maybe) one for the buffers.
         *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         * This isn't perfect, but works for just about everything.
         * Besides, as long as we don't move unfreeable pages to the
         * inactive_clean list it doesn't need to be perfect...
         */
        int maxcount = (page->buffers ? 3 : 2);
        page->age = 0;
        ClearPageReferenced(page);

        /*
         * Don't touch it if it's not on the active list.
         * (some pages aren't on any list at all)
         */
        if (PageActive(page) && page_count(page) <= maxcount && !page_ramdisk(page)) {
             del_page_from_active_list(page);
             add_page_to_inactive_dirty_list(page);
        }
}
```

在物理页面的 page 结构中有个计数器 count，空闲页面的这个计数为 0，在分配页面时将其设为 1（见__alloc_pages()和 rmqueue()的代码），此后每当页面增加一个“用户”，如建立或恢复一个映射时，就使 count 加 1。这样，如果这个计数器的值为 2，就说明刚断开的映射已经是该物理页面的最后一个映射。既然最后的映射已经断开，这页面当然是不活跃的了。所以把小于等于 2 作为一个判断的准则，就是这里的 maxcount。但是，这里还要考虑一种特殊情况，就是当这个页面是通过 mmap()映射到普通文件，而这个文件又已经被打开，按常规的文件操作访问，因此这个页面又同时用作读 / 写文件的缓冲。此时页面划分成若干缓冲区，其 page 结构中的指针 buffers 指向一个 buffer_head 数据结构队列，而这个队列则成了该页面的另一个“用户”。所以，当 page->buffers 非 0 时，maxcount 为 3 说明刚断开的映射是该内存页面的最后一个映射。此外，内存页面也有可能用作 ramdisk，即以一部分内存物理空间来模拟硬盘，这样的页面永远不会变成不活跃。这样，判断的准则一共有三条，只有在满足了这三条准则时才真的可以将页面转入不活跃队列。多数有用户空间映射的内存页面都只有一个映射，此时就转入了不活跃状态。同时，从代码中也可看出，对不在活跃队列中的页面再调用一次 deactivate_page_nolock()并无害处。

将一个活跃的页面变成不活跃时，要把该页面的 page 结构从活跃页面的 LRU 队列 active_list 中转移到一个不活跃队列中去。可是，系统中有两种不活跃页面队列。一种是“ dirty”，即可能最近已被写过，因而跟交换设备上的页面不一致的“脏”页面队列，这样的页面不能马上就拿来分配，因为还需要把它写出去才能把它“洗净”。另一种是“ clean”，即肯定跟交换设备上的页面一致的“干净”页面队列，这样的页面原则上已可作为空闲页面分配，只是因为页面中的内容还可能有用，因而再予以保存一段时间。不活跃“脏”页面队列只有一个，那就是 inactive_dirty_list；而不活跃“干净”页面队列则有很多，每个页面管理区中都有个 inactive_clean_list 队列。那么，当一个原来活跃的页面变成不活跃时，应该把它转移到哪一个队列中去呢？第一步总是把它转入“脏”页面队列。将一个 page 结构从活跃队列脱链是由宏操作 del_page_from_active_list()完成的，其定义在 include/linux/swap.h 中：

==================== include/linux/swap.h 234 240 ====================

```
#define del_page_from_active_list(page) { \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    list_del(&(page)->lru); \
    ClearPageActive(page); \
    nr_active_pages--; \
    DEBUG_ADD_PAGE \
    ZERO_PAGE_BUG \
}
```

将一个 page 结构链入不活跃队列，则由 add_page_to_inactive_dirty_list()完成：

==================== include/linux/swap.h 217 224 ====================

```
#define add_page_to_inactive_dirty_list(page) { \
    DEBUG_ADD_PAGE \
    ZERO_PAGE_BUG \
    SetPageInactiveDirty(page); \
    list_add(&(page)->lru, &inactive_dirty_list); \
    nr_inactive_dirty_pages++; \
    page->zone->inactive_dirty_pages++; \
}
```

这里的 ClearPageActive()和 SetPageInactiveDirty()分别将 page 结构中的 PG_active 标志位清成 0 和将 PG_inactive_dirty 标志位设成 1。注意在这个过程中 page 结构中的使用计数并未改变。

又回到 try_to_swap_out()的代码中，既然断开了对一个内存页面的映射，就要递减对这个页面的使用计数，这是由宏操作 page_cache_release()、实际上是由_free_pages()完成的。

==================== include/linux/pagemap.h 34 34 ====================

```
#define page_cache_release(x)    __free_page(x)
```

==================== include/linux/mm.h 379 379 ====================

```
#define __free_page(page) __free_pages((page), 0)
```

==================== mm/page_alloc.c 549 553 ====================

```
void __free_pages(struct page *page, unsigned long order)
{
    if (!PageReserved(page) && put_page_testzero(page))
        __free_pages_ok(page, order);
}
```

==================== include/linux/mm.h 152 152 ====================

```
#define put_page_testzero(p)    atomic_dec_and_test(&(p)->count)
```

这个函数通过 put_page_testzero()，将 page 结构中 count 的值减 1，然后测试是否达到了 0，如果达到了 0 就通过__free_pages_ok()将该页面释放。在这里，由于页面还在不活跃页面队列中尚未释放，至少还有这么一个引用，所以不会达到 0。

至此，对这个页面的处理就完成了，于是又到了标号 out_failed 处而返回 0。为什么又是到达 out_failed 处呢？其实，try_to_swap_out()仅在一种情况下才返回 1，那就是当 mm->swap_cnt 达到了 0 的时候（见 52 行）。正是这样，才使 swap_out_mm()能够依次考察和处理一个进程的所有页面。

要是页面的 page 结构不在 swapper_space 的队列中呢？这说明尚未为该页面在交换设备上建立起映象，或者页面来自一个文件。读者可以回顾一下，在因页面无映射而发生缺页异常时，具体的处理取决于页面所在的区间是否提供了一个 vm_operations_struct 数据结构，并且通过这个数据结构中的函

PDF-XCHANGE Click to buy NOW! www.docu-track.com

数指针 nopage 提供了特定的操作。如果提供了 nopage 操作，就说明该区间的页面来自一个文间（而不是交换设备），此时根据虚存地址时以计算出在文件中的页面位置。否则就是普通的页面，但尚未建立相应的盘上页面（因为页面表项为 0），此时先把它映射到空白页面，以后需要写的时候才为之另行分配一个页面。我们继续往下看 try_to_swap_out()的代码，下面一段就是对此种页面的处理：

==================== mm/vmscan.c 110 157 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()]

```
110        /*
111         * Is it a clean page? Then it must be recoverable
112         * by just paging it in again, and we can just drop
113         * it..
114         *
115         * However, this won't actually free any real
116         * memory, as the page will just be in the page cache
117         * somewhere, and as such we should just continue
118         * our scan.
119         *
120         * Basically, this just makes it possible for us to do
121         * some real work in the future in "refill_inactive()".
122         */
123        flush_cache_page(vma, address);
124        if (!pte_dirty(pte))
125             goto drop_pte;
126
127        /*
128         * Ok, it's really dirty. That means that
129         * we should either create a new swap cache
130         * entry for it, or we should write it back
131         * to its own backing store.
132         */
133        if (page->mapping) {
134             set_page_dirty(page);
135             goto drop_pte;
136        }
137
138        /*
139         * This is a dirty, swappable page.  First of all,
140         * get a suitable swap entry for it, and make sure
141         * we have the swap cache set up to associate the
142         * page with that swap entry.
143         */
144        entry = get_swap_page();
145        if (!entry.val)
146             goto out_unlock_restore; /* No swap space left */
147
148        /* Add it to the swap cache and mark it dirty */
149        add_to_swap_cache(page, entry);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
150     set_page_dirty(page);
151     goto set_swap_pte;
152
153 out_unlock_restore:
154     set_pte(page_table, pte);
155     UnlockPage(page);
156     return 0;
157 }
```

这里的 pte_dirty()是一个 inline 函数，定义于 include/asm-i386/pgtable.h：

==================== include/asm-i386/pgtable.h 269 269 ====================

```
269 static inline int pte_dirty(pte_t pte)      { return (pte).pte_low & _PAGE_DIRTY; }
```

在页面表项中有一个“D”标志位（_PAGE_DIRTY），如果 CPU 对表项所指的内存页面进行了写操作，就自动把该标志位设置成 1，表示该内存页面已经“脏”了。如果此标志位为 0，就表示相应的内存页面尚未被写过。对这样的页面，如果很久没有受到写访问，就可以把映射解除（而不是暂时断开）。这是因为；如果页面的内容是空白，那么以后需要时可以再来建立映射；或者，如果页面来自通过 mmap()建立起的文件映射，则在需要时可以根据虚拟地址计算出页面在文件中的位置（相比之下，交换设备上的页面位置不能通过计算得到，所以必须把页面的去向存储在页面表项中）。所以，这里转到前面的标号 drop_pte 处。注意在这种情况下前面的 deactivate_pape()实际上不起作用，特别是页面表项已在前面 83 行清 0，而 page_cache_release()则只是递减对空白页面的引用计数。

如果所考察的页面是来自通过 mmap()建立起的文件映射，则其 pape 结构中的指针 mapping 指向相应的 address_space 数据结构。对于这样的页面，如果决定解除映射，而页面表项中的_PAGE_DIRTY 标志位为 1，就要在转到 drop_pte 处之前，先把 page 结构中的 PG_dirty 标志位设成 1，并把页面转移到该文件映射的“脏”页面队列中。有关的操作 set_page_dirty()定义于 include/linux/mm.h 以及 mm/filemap.c

==================== include/linux/mm.h 187 191 ====================

[kswapd()>_do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()>set_page_dirty()]

```
187 static inline void set_page_dirty(struct page * page)
188 {
189     if (!test_and_set_bit(PG_dirty, &page->flags))
190         __set_page_dirty(page);
191 }
```

==================== mm/filemap.c 134 147 ====================

```
134 /*
135  * Add a page to the dirty page list.
136  */
137 void __set_page_dirty(struct page *page)
138 {
139     struct address_space *mapping = page->mapping;
140
141     spin_lock(&pagecache_lock);
```

```
142         list_del(&page->list);
143         list_add(&page->list, &mapping->dirty_pages);
144         spin_unlock(&pagecache_lock);
145
146         mark_inode_dirty_pages(mapping->host);
147  }
```

再往下看 try_to_swap_out()的代码。当程序执行到这里时，所考察的页面必然是个很久没有受到访问，又不在 swapper_space 的换入／换出队列中，也不属于文件映射，但却是个受到过写访问的“脏”页面。对于这样的页面必须要为之分配一个盘上页面，并将其内容写到盘上页面中去。首先通过 get_swap_page()分配一个盘上页面，这是个宏操作：

==================== include/linux/swap.h 150 150 ====================

```
150  #define get_swap_page() __get_swap_page(1)
```

就是说，通过__get_swap_page()从交换设备上分配一个页面。其代码在 mm/swapfile.c 中，由于比较简单，我们把它留给读者。盘上页面的使用计数在分配时设置成 1，以后每当有进程参与共享同一内存页面时就通过 swap_duplicate()递增，此外在有进程断开对此页面的映射时也要递增（见 99 行）；反之则通过 swap_free()递减。如果分配盘上页面失败，就转到 out_unlock_restore 处恢复原有的映射。

分配了盘上页面以后，就通过 add_to_swap_cache()将页面链入 swapper_space 的队列中，以及活跃页面队列中，这个函数的代码以前已经看到过了。然后，再通过 set_page_dirty()将页面转到不活跃“脏”页面队列中。至于实际的写出，则前面已经看到是 page_launder()的事。

至此，对一个进程的用户空间页面的扫描处理就完成了。swap_out()是在一个 for 循环中调用 swap_out_mm()的，所以每次调用 swap_out()都会换出若干进程的若干页面，而 refill_inactive()又是在嵌套的 while 循环中调用 swap_out()的，一直要到系统中可供分配的页面，包括潜在可供分配的页面在内不再短缺时为止。到那时，do_try_to_free_pages()就结束了。回到 kswapd()的代码中，此时活跃页面队列的情况可能已经有了较大的改变，所以还要再调用一次 refill_inactive_scan()。这样，kswapd()的一次例行路线就基本走完了。如前所述，kswapd()除定期的执行外，也有可能是被其它进程唤醒的，所以可能有进程正在睡眠中等待其完成，因此通过 wake_up_all()唤醒这些进程。

读者也许在想，通过 swap_out_mm()对每个进程页面表的扫描并不保证一定能有页面转入不活跃状态，这样 refill_inactive()岂不是要无穷无尽地循环下去？事实上，一来程序中对循环的次数有个限制，二来对页面表的扫描是个自适应的过程。如果在对所有进程的一轮扫描后转入不活跃状态的页面数量不足；那么 refill_inactive()就会又回过头来开始第二轮扫描。而扫描次数的增加会使页面老化的速度也增加，因为页面的寿命实际上是以扫描的次数为单位的。这样，在第一轮扫描中不符合条件的页面在第二轮扫描中就可能符合条件了。最后，在很特殊的情况下，可能最终还是达不到要求，此时就调用 oom_kill()从系统中杀掉一个进程，通过牺牲局部来保障全局。

最后，再来看看线程 kreclaimd 的代码，这是在 mm/vmscan.c 中：

==================== mm/vmscan.c 1095 1143 ====================

```
1095 DECLARE_WAIT_QUEUE_HEAD(kreclaimd_wait);
1096 /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Kreclaimd will move pages from the inactive_clean list to the
 * free list, in order to keep atomic allocations possible under
 * all circumstances. Even when kswapd is blocked on IO.
 */
int kreclaimd(void *unused)
{
	struct task_struct *tsk = current;
	pg_data_t *pgdat;

	tsk->session = 1;
	tsk->pgrp = 1;
	strcpy(tsk->comm, "kreclaimd");
	sigfillset(&tsk->blocked);
	current->flags |= PF_MEMALLOC;

	while (1) {

		/*
		 * We sleep until someone wakes us up from
		 * page_alloc.c::__alloc_pages().
		 */
		interruptible_sleep_on(&kreclaimd_wait);

		/*
		 * Move some pages from the inactive_clean lists to
		 * the free lists, if it is needed.
		 */
		pgdat = pgdat_list;
		do {
			int i;
			for(i = 0; i < MAX_NR_ZONES; i++) {
				zone_t *zone = pgdat->node_zones + i;
				if (!zone->size)
					continue;

				while (zone->free_pages < zone->pages_low) {
					struct page * page;
					page = reclaim_page(zone);
					if (!page)
						break;
					__free_page(page);
				}
			}
			pgdat = pgdat->node_next;
		} while (pgdat);
	}
}
```

对照一下 kswapd()的代码，就可以看出二者的初始化部分是一样的，程序的结构也相似。注意二

PDF-XCHANGE Click to buy NOW! www.docu-track.com

者都把其 task_struct 结构中 flags 字段的 PF_MEMALLOC 标志位设成 1，表示这两个内核线程都是页面管理机制的维护者。事实上，在以前的版本中只有一个线程 kswapd，在 2.4 版中才把其中的一部分独立出来成为一个线程。不过，这一次是通过 reclaim_page()扫描各个页面管理区中的不活跃“干净”页面队列；从中回收页面加以释放。这个函数的代码在 mm/vmscan.c 中，我们把它留给读者自己阅读。在阅读了上面这些代码以后，读者已经不至于感到困难了。

==================== mm/vmscan.c 381 463 ====================

[kreclaimd()>reclaim_page()]

```
/**
 * reclaim_page -  reclaims one page from the inactive_clean list
 * @zone: reclaim a page from this zone
 *
 * The pages on the inactive_clean can be instantly reclaimed.
 * The tests look impressive, but most of the time we'll grab
 * the first page of the list and exit successfully.
 */
struct page * reclaim_page(zone_t * zone)
{
     struct page * page = NULL;
     struct list_head * page_lru;
     int maxscan;

     /*
      * We only need the pagemap_lru_lock if we don't reclaim the page,
      * but we have to grab the pagecache_lock before the pagemap_lru_lock
      * to avoid deadlocks and most of the time we'll succeed anyway.
      */
     spin_lock(&pagecache_lock);
     spin_lock(&pagemap_lru_lock);
     maxscan = zone->inactive_clean_pages;
     while ((page_lru = zone->inactive_clean_list.prev) !=
               &zone->inactive_clean_list && maxscan--) {
          page = list_entry(page_lru, struct page, lru);

          /* Wrong page on list?! (list corruption, should not happen) */
          if (!PageInactiveClean(page)) {
               printk("VM: reclaim_page, wrong page on list.\n");
               list_del(page_lru);
               page->zone->inactive_clean_pages--;
               continue;
          }

          /* Page is or was in use?  Move it to the active list. */
          if (PageTestandClearReferenced(page) || page->age > 0 ||
                    (!page->buffers && page_count(page) > 1)) {
               del_page_from_inactive_clean_list(page);
               add_page_to_active_list(page);
               continue;
```

```
        }

        /* The page is dirty, or locked, move to inactive_dirty list. */
        if (page->buffers || PageDirty(page) || TryLockPage(page)) {
            del_page_from_inactive_clean_list(page);
            add_page_to_inactive_dirty_list(page);
            continue;
        }

        /* OK, remove the page from the caches. */
            if (PageSwapCache(page)) {
            __delete_from_swap_cache(page);
            goto found_page;
        }

        if (page->mapping) {
            __remove_inode_page(page);
            goto found_page;
        }

        /* We should never ever get here. */
        printk(KERN_ERR "VM: reclaim_page, found unknown page\n");
        list_del(page_lru);
        zone->inactive_clean_pages--;
        UnlockPage(page);
    }
    /* Reset page pointer, maybe we encountered an unfreeable page. */
    page = NULL;
    goto out;

found_page:
    del_page_from_inactive_clean_list(page);
    UnlockPage(page);
    page->age = PAGE_AGE_START;
    if (page_count(page) != 1)
        printk("VM: reclaim_page, found page with count %d!\n",
                page_count(page));
out:
    spin_unlock(&pagemap_lru_lock);
    spin_unlock(&pagecache_lock);
    memory_pressure++;
    return page;
}
```

2.9 页面的换入

在 i386 CPU 将一个线性地址映射成物理地址的过程中。如果该地址的映射已经建立，但还发现相

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

应页面表项或目录项中的 P（Present）标志位为 0，则表示相应的物理页面不在内存，从而无法完成本次内存访问。从理论上说，也许应该把这种情况称为“受阻”而不是“失败”，因为映射的关系毕竟已经建立，理应与尚未建立映射的情况有所区别，所以我们称之为“断开”。但是，CPU 的 MMU 硬件并不区分这两种不同的情况，只要 P 标志位为 0 就都认为是页面映射失败，CPU 就会产生一次“页面异常”（Page Fault）。事实上，CPU 在映射过程中首先看的就是页面表项或目录项中的 P 标志位。只要 P 标志位为 0，其余各个位段的值就无意义了。至于当一个页面不在内存中时，利用页面表项指向一个盘上页面，那是软件的事。所以，区分失败的原因到底是因为页面不在内存，还是因为映射尚未建立，乃是软件，也就是页面异常处理程序的事。在“越界访问”的情景中，我们曾看到在函数 handle_pte_fault()中的开头几行：

==================== mm/memory.c 1153 1175 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()]

```
static inline int handle_pte_fault(struct mm_struct *mm,
	struct vm_area_struct * vma, unsigned long address,
	int write_access, pte_t * pte)
{
	pte_t entry;

	/*
	 * We need the page table lock to synchronize with kswapd
	 * and the SMP-safe atomic PTE updates.
	 */
	spin_lock(&mm->page_table_lock);
	entry = *pte;
	if (!pte_present(entry)) {
		/*
		 * If it truly wasn't present, we know that kswapd
		 * and the PTE updates will not touch it later. So
		 * drop the lock.
		 */
		spin_unlock(&mm->page_table_lock);
		if (pte_none(entry))
			return do_no_page(mm, vma, address, write_access, pte);
		return do_swap_page(mm, vma, address, pte, pte_to_swp_entry(entry), write_access);
	}
```

这里，首先区分的是 pte_present()，也就是检查表项中的 P 标志位，看看物理页面是否在内存中。如果不在，则进而通过 pte_none()检查表项是否为空，即全 0。如果为空就说明映射尚未建立，所以要调用 do_no_page()。这在以前的情景中已经看到过了。反之，如果非空，就说明映射已经建立，只是物理页面不在内存中，所以要通过 do_swap_page()，从交换设备上换入这个页面。本情景在 handle_pte_fault()之前的处理以及执行路线都与越界访问的情景相同，所以我们直接进入 do_swap_page()。这个函数的代码在 mm/memory.c 中：

==================== mm/memory.c 1018 1056 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_swap_page()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static int do_swap_page(struct mm_struct * mm,
        struct vm_area_struct * vma, unsigned long address,
        pte_t * page_table, swp_entry_t entry, int write_access)
{
        struct page *page = lookup_swap_cache(entry);
        pte_t pte;

        if (!page) {
                lock_kernel();
                swapin_readahead(entry);
                page = read_swap_cache(entry);
                unlock_kernel();
                if (!page)
                        return -1;

                flush_page_to_ram(page);
                flush_icache_page(vma, page);
        }

        mm->rss++;

        pte = mk_pte(page, vma->vm_page_prot);

        /*
         * Freeze the "shared"ness of the page, ie page_count + swap_count.
         * Must lock page before transferring our swap count to already
         * obtained page count.
         */
        lock_page(page);
        swap_free(entry);
        if (write_access && !is_page_shared(page))
                pte = pte_mkwrite(pte_mkdirty(pte));
        UnlockPage(page);

        set_pte(page_table, pte);
        /* No need to invalidate - it was non-present before */
        update_mmu_cache(vma, address, pte);
        return 1;/* Minor fault */
}
```

先看看调用时传过来的参数是些什么。建议读者先回到前面通过越界访问扩充堆栈的情景中，顺着 CPU 的执行路线走一遍，搞清楚这些参数的来龙去脉。参数表中的 mm、vma 还有 address 是一目了然的，分别是指向当前进程的 mm_struct 结构的指针、所属虚存区间的 vm_area_struct 结构的指针以及映射失败的线性地址。

参数 page_table 指向映射失败的页面表项，而 entry 则为该表项的内容。我们以前说过，当物理页面在内存中时，页面表项是一个叫 pte_t 结构，指向一个内存页面；而当物理页面不在内存中时，则是一个 swap_entry_t 结构，指向一个盘上页面。二者实际上都是 32 位无符号整数。这里要指出，所谓“不

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在内存中”是逻辑意义上的，是对 CPU 的页面映射硬件而言，实际上这个页面很可能在不活跃页面队列中，甚至在活跃页面队列中。

还有一个参数 write_access，表示当映射失败时所进行的访问种类（读 / 写），这是在 do_page_fault()的 switch 语句中（见 arch/i386/fault.c）根据 CPU 产生的出错代码 error_code 的 bit1 决定的（注意，在那个 switch 语句中，“ default:”与“ case 2:”之间没有 break 语句）。此后便逐层传了下来。

由于物理页面不在内存，所以 entry 是指向一个盘上页面的类似于指针的索引项（加上若干标志位）。该指针逻辑上分成两部分：第一部分是页面交换设备（或文件）的序号；第二部分是页面在这个设备上（或文件中，下同）的位移，其实也就是页面序号。两部分合在一起就惟一地确定了一个盘上页面。供页面交换的设备上第一个页面（序号为 0）是保留不用的，所以 entry 的值不可能为全 0。这样才能与映射尚未建立时的页面表项相区别。

处理一次因缺页面引起的页面异常时，首先要看看相应的内存页面是否还留在 swapper_space 的换入 / 换出队列中尚未最后释放。如果是的话那就省事了。所以，要先调用 lookup_swap_cache()。这个函数是在 swap_state.c 中定义的，我们把它留给读者自己阅读。

如果没有找到，就是说以前用于这个虚存页面的内存页面已经释放，现在其内容仅存在于盘上了，那就要通过 read_swap_cache()分配一个内存页面，并且从盘上将其内容读进来。为什么在此之前要先调用 swapin_readahead()呢？当从磁盘上读的时候，每次仅仅读一个页面是不经济的，因为每次读盘都要经过在磁盘上寻道使磁头定位，而寻道所需的时间实际上比磁头到位以后读一个页面所需的时间要长得多。所以，比较经济的办法是：既然必须经过寻道，就干脆一次多读几个页面进来，称为一个页面集群（cluster）。由于此时并非每个读入的页面都是立即需要的，所以是“预读”（read ahead）。预读进来的页面都暂时链入活跃页面队列以及 swapper_space 的换入 / 换出队列中，如果实际上确实不需要就会由进程 kswapd 和 kreclaimd 在一段时间以后加以回收。这样，当调用 read_swap_cache()时，通常所需的页面已经在活跃页面队列中而只需要把它找到就行了。但是，也有可能预读时因为分配不到足够的内存页面而失败，那样就真的要再来读一次，而这一次却真是只读入一个页面了。细心的读者可能会问，这两行程序是紧挨着的，为什么在前一行语句中因分配不到足够的内存页面而失败，到紧接着的下一行就有可能成功呢？这是因为，在分配内存页面失败时，内核可能会调度其它进程先运行，而被调度运行的进程可能会释放出一些内存页面，甚至被调度运行的进程可能恰好就是 kswapd。因此，第一次分配内存页面失败并不一定说明紧接着的第二次也会失败。要明白这一点，我们可以再来看一下函数__alloc_pages()中的一个片段：

==================== mm/page_alloc.c 382 387 ====================

```
	wakeup_kswapd(0);
	if (gfp_mask & __GFP_WAIT) {
		__set_current_state(TASK_RUNNING);
		current->policy |= SCHED_YIELD;
		schedule();
	}
```

无论是 swapin_readahead()还是 read_swap_cache()，在申请分配内存页面时都把调用参数 gfp_mask 中的__GFP_WAIT 标志位置成 1，所以当分配不到内存页面时都会自愿暂时礼让，让内核调度其它进程先运行。由于在此之前先唤醒了 kswapd，当本进程被调度恢复运行时，也就是从 schedule()返回时，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

再次试图分配页面已有可能成功了。即使在 swapin_readahead()中又失败了，在 read_swap_cache()中再来一次，也还是有可能（而且多半能够）成功。当然，也有可能二者都失败了，那样 do_swap_page()也就失败了，所以在 1031 行返回-1。这里，我们就不深入到 swapin_readahead()中去了，读者可以自行阅读。而 read_swap_cache()实际上是 read_swap_cache_async()，只是把调用参数 wait 设成 1，表示要等待读入完成（所以实际上是同步的读入）。

==================== include/linux/swap.h 125 125 ====================

```
#define read_swap_cache(entry) read_swap_cache_async(entry, 1);
```

函数 read_swap_cache_async()的代码在 mm/swap_state.c 中：

==================== mm/swap_state.c 204 255 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_swap_page()read_swap_cache_async()]

```
/*
 * Locate a page of swap in physical memory, reserving swap cache space
 * and reading the disk if it is not already cached.  If wait==0, we are
 * only doing readahead, so don't worry if the page is already locked.
 *
 * A failure return means that either the page allocation failed or that
 * the swap entry is no longer in use.
 */

struct page * read_swap_cache_async(swp_entry_t entry, int wait)
{
     struct page *found_page = 0, *new_page;
     unsigned long new_page_addr;

     /*
      * Make sure the swap entry is still in use.
      */
     if (!swap_duplicate(entry)) /* Account for the swap cache */
          goto out;
     /*
      * Look for the page in the swap cache.
      */
     found_page = lookup_swap_cache(entry);
     if (found_page)
          goto out_free_swap;

     new_page_addr = __get_free_page(GFP_USER);
     if (!new_page_addr)
          goto out_free_swap;     /* Out of memory */
     new_page = virt_to_page(new_page_addr);

     /*
      * Check the swap cache again, in case we stalled above.
      */
     found_page = lookup_swap_cache(entry);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
239         if (found_page)
240              goto out_free_page;
241         /*
242          * Add it to the swap cache and read its contents.
243          */
244         lock_page(new_page);
245         add_to_swap_cache(new_page, entry);
246         rw_swap_page(READ, new_page, wait);
247         return new_page;
248
249    out_free_page:
250         page_cache_release(new_page);
251    out_free_swap:
252         swap_free(entry);
253    out:
254         return found_page;
255    }
```

读者也许注意到了，这里两次调用了 lookup_swap_cache()。第一次是很好理解的，因为 swapin_readahead()也许已经把目标页面读进来了，所以要先从 swapper_space 队列中寻找一次。这一方面是为了节省一次从设备读入；另一方面，更重要的是防止同一个页面在内存中有两个副本。可是为什么在找不到、因而为此分配了一个内存页面以后又来寻找一次呢？这是因为分配内存页面的过程有可能受阻，如果一时分配不到页面，当前进程就会睡眠等待，让别的进程先运行。而当这个进程再次被调度运行，并成功地分配到物理页面从__get_free_page()返回时，也许另一个进程已经先把这个页面读进来了，所以要再检查一次。如果确实需要从交换设备读入，则通过 add_to_swap_cache()将新分配的物理页面（确切地说是它的 page 数据结构）挂入 swapper_space 队列以及 active_list 队列中，这个函数的代码读者已经看到过了。至于 rw_swap_page()，读者可以在学习了块设备驱动一章以后回过来阅读。调用 read_swap_cache()成功以后，所要的页面肯定已经在 swapper_space 队列以及 active_list 队列中了，并且马上就要恢复映射。

这里要着重注意一下对盘上页面的共享计数。首先，一开始时在 221 行就通过 swap_duplicate()递增了盘上页面的共享计数。如果在缓冲队列中找到了所需的页面而无需从交换设备读入，则在 252 行通过 swap_free()抵消对共享计数的递增。反之，如果需要从交换设备读入页面，则不调用 swap_free()，所以盘上页面的共享计数加了 1。可是，回到 do_swap_page()以后，在 1047 行又调用了一次 swap_free()，使盘上页面的共享计数减 1。这么一来，情况就变成了这样：如果从交换设备读入页面，则盘上页面的共享计数保持不变；而如果在缓冲队列中找到了所需的页面，则共享计数减 1。对此，读者不妨回过去看一下 try_to_swap_out()中的 99 行。在那里，当断开一个页面的映射时，通过 swap_duplicate()递增了盘上页面的共享计数。而现在恢复映射则使共享计数减 1，二者是互相对应的。

还要注意对内存页面，即其 page 结构的使用计数。首先，在分配一个内存页面时把这个计数设成 1。然后，在通过 add_to_swap_cache()将其链入换入／换出队列（或文件映射队列）和 LRU 队列 active_list 时，又在 add_to_page_cache_locked()中通过 page_cache_get()递增了这个计数，所以当有、并且只有一个进程映射到这个换入／换出页面时，其使用计数为 2。如果页面来自文件映射，则由于同时又与文件读／写缓冲区相联系，又多一个“用户”，所以使用计数为 3。但是，还有一种特殊情况，那就是通过

swapin_readahead()预读进来的页面。

==================== mm/memory.c 990 991 ====================
[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_swap_page()>swapin_readahead()]

```
void swapin_readahead(swp_entry_t entry)
{
  ......
```

==================== mm/memory.c 1001 1001 ====================

```
    for (i = 0; i < num; offset++, i++) {
  ......
```

==================== mm/memory.c 1009 1016 ====================

```
        /* Ok, do the async read-ahead now */
        new_page = read_swap_cache_async(SWP_ENTRY(SWP_TYPE(entry), offset), 0);
        if (new_page != NULL)
            page_cache_release(new_page);
        swap_free(SWP_ENTRY(SWP_TYPE(entry), offset));
    }
    return;
}
```

在 swapin_readahead()中，循环地调用 read_swap_cache_async()分配和读入若干页面，因而在从 read_swap_cache_async()返回时，每个页面的使用计数都是 2。但是，在循环中马上又通过 page_cache_release()递减这个计数，因为预读进来的页面并没有进程在使用。于是，这些页面就成了特殊的页面，它们在 active_list 中，而使用计数却是 1。以后，这些页面或者是被某个进程"认领"，从而使用计数变成 2；或者是在一段时间以后仍无进程认领，最后被 refill_inactive_scan()移入不活跃队列（见 mm/vmscan.c 的 744 行），那才是使用计数为 1 的页面应该呆的地方。

回到 do_swap_page()代码中，这里的 flush_page_to_ram()和 flush_icache_page()对于 i386 处理器均为空操作。代码中通过 pte_mkdirty()将页面表项中的 D 标志位置成 1，表示该页面已经"脏"了，并且通过 pte_mkwrite()将页面表项中的_PAGE_RW 标志位也置成 1。读者也许会问：怎么可以凭着当前的访问是一次写访问就把页面表项设置成允许写？万一本来就应该有写保护的呢？答案是，如果那样的话就根本到达不了这个地方。读者不妨回过头去看看 do_page_fault()中 switch 语句的 case 2。在那用，如果页面所属的区间不允许写的话（VM_WRITE 标志位为 0），就转到 bad_area 去了。还要注意，区间的可写标志 VM_WRITE 与页面的可写标志_PAGE_RW 是不同的。VM_WRITE 是个相对静态的标志位；而_PAGE_RW 则更为动态，只表示当前这一个物理内存页面是否允许写访问。只有在 VM_WRITE 为 1 的前提下，_PAGE_RW 才有可能为 1，但却并不一定为 1。所以，在 1039 行中，根据 vma->vm_page_prot 构筑一个页面表项时，表项的_PAGE_RW 标志位为 0（注意 VM_WRITE 是 vma->vm_flags 而不是 vma->vm_page_prot 中的一位）。读者还可能会问，那样一来，要是当前的访问恰好是读访问，这个页面不就永远不允许写了吗？不要紧，发生写访问时会因访问权限不符而引起另一次页面异常。那时，就会在 handle_pte_fault()中调用 do_wp_page()，将页面的访问权限作出改变（如果需要 cow，即 copy_on_write 的话，也是在那里处理的）。我们将 do_wp_page()留给读者，一来是因为篇幅的关系，二来读者现在对存储管理已经比较熟悉，应该不会有太大的困难了。

至于紧接着的 update_mmu_cache()，对于 i386 CPU 只是个空操作，因为 i386 的 MMU 是与 CPU 汇成一体的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

2.10 内核缓冲区的管理

可想而知，内核在运行中常常会需要使用一些缓冲区。例如，当要建立一个新的进程时就要增加一个 task_struct 结构，而当进程撤销时就要释放本进程的 task 结构。这些小块存储空间的使用并不局限于某一个子程序，否则就可以作为这个子程序的局部变量而使用堆栈空间了。另外，这些小块存储空间又是动态变化的，不像用于内存页面管理的 page 结构那样，有多大的内存就有多少个 page 结构，构成一个静态的阵列。由于事先根本无法预测运行中各种不同数据结构对缓冲区的需求，不适合为每一种可能用到的数据结构建立一个"缓冲池"，因为那样的话很可能会出现有些缓冲池已经用尽而有些缓冲池中却有大量缓冲区空闲的局面。因此，只能采用更具全局性的方法。

那么，用什么样的方法呢？如果采用像用户空间中的 malloc()那样的动态分配办法，从一个统一的存储空间"堆"（heap）中，需要用多少就切下多大一块，不用了就归还，则有几个缺点需要考虑改进：

- 久而久之，会使存储堆"碎片化"，以至虽然存储堆中空闲空间的总量足够大、却无法分配所需大小的连续空间。为此，一般都采用按 2^n 的大小来分配空间，以缓解碎片化。
- 每次分配得到所需大小的缓冲区以后，都要进行初始化。内核中频繁地使用一些数据结构，这些数据结构中相当一部分成分需要某些特殊的初始化（例如队列头部等）而并非简单地清成全 0。如果释放的数据结构可以在下次分配时"重用"而无需初始化，那就可以提高内核的效率。
- 缓冲区的组织和管理是密切相关的。在有高速缓存的情况下，这些缓冲区的组织和管理方式直接影响到高速缓存中的命中率，进而影响到运行时的效率。试想，假定我们运用最先符合（first fit）的方法，从一个由存储空间片段构成的队列中分配缓冲区。在这样的过程中，当一个片段不能满足要求而顺着指针往下看下一个片段的数据结构时，如果该数据结构每次都在不同的页面中，因而每次都不能命中，而要从内存装入到高速缓存，那么可想而知，其效率显然就要打折扣了。
- 不适合多处理器共用内存的情况。

实际上，如何有效地管理缓冲区空间，很久以来就是一个热门的研究课题。90 年代前期，在 Solaris 2.4 操作系统（Unix 的一个变种）中，采用了一种称为" slab"的缓冲区分配和管理方法（slab 的原意是大块的混凝土），在相当程度上克服了上述的缺点。而 Linux，也在其内核中采用了这种方法，并作了改进。

从存储器分配的角度讲，slab 与为各种数据结构分别建立缓冲池相似；也与以前我们看到过的按大小划分管理区（zone）的方法相似，但是也有重要的不同。

在 slab 方法中，每种重要的数据结构都有自己专用的缓冲区队列，每种数据结构都有相应的"构造"（constructor）和"拆除"（destructor）函数。同时，还借用面向对象程序设计技术中的名词，不再称"结构"而称为"对象"（object）。缓冲区队列中的各个对象在建立时用其"构造"函数进行初始化，所以一经分配立即就能使用，而在释放时则恢复成原状。例如，对于其中的队列头成分来说（读者可参看 page 数据结构的定义，结构中有两个 struct list_head 成分），当将其从队列中摘除时自然就恢复成了原状。每个队列中"对象"的个数是动态变化的，不够时可以增添。同时，又定期地检查，将有富余的队列加以精简。我们在 kswapd 的 do_try_to_free_pages()中曾经看到，调用函数 kmem_cache_reap()，

为的就是从富余的队列回收物理页面，只是当时我们没有细讲。其实，定期地检查和处理这些缓冲区队列，也是 kswapd 的一项功能。

此外，slab 管理方法还有一个特点，每种对象的缓冲区队列并非由各个对象直接构成，而是由一连串的“大块”（slab）构成，而每个大块中则包含了若干同种的对象。一般而言，对象分两种，一种是大对象，一种是小对象。所谓小对象，是指在一个页面中可以容纳下好几个对象的那一种。例如，一个 inode 的大小约 300 多个字节，因此一个页面中可以容纳 8 个以上的 inode，所以 inode 是小对象。内核中使用的大多数数据结构都是这样的小对象，所以，我们先来看对小对象的组织和管理以及相应的 slab 结构。先看用于某种假想小对象的一个 slab 块的结构示意图（图 2.8）：

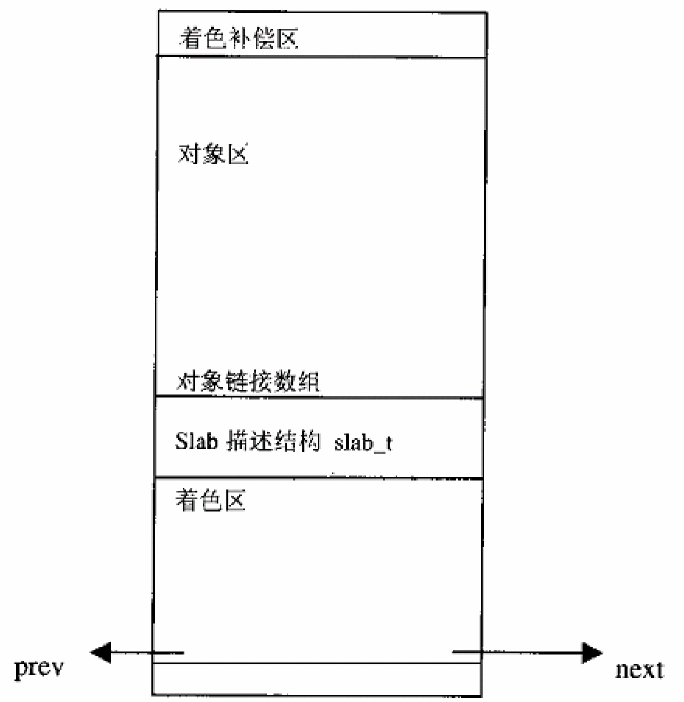


图 2.8　小对象 slab 结构示意图

此处先对上述示意图作几点说明，详细情况则随着代码的阅读再逐步深入；

- 一个 slab 可能由 1 个、2 个、4 个、…最多 32 个连续的物理页面构成。slab 的具体大小因对象的大小而异，初始化时通过计算得出最合适的大小。
- 在每个 slab 的前端是该 slab 的描述结构 slab_t。用于同一种对象的多个 slab 通过描述结构中的队列头形成一条双向链队列。每个 slab 双向链队列在逻辑上分成三截，第一截是各个 slab 上所有的对象都已分配使用的；第二截是各个 slab 上的对象已经部分地分配使用；最后一截是各个 slab 上的全部对象都处于空闲状态。
- 每个 slab 都有一个对象区，这是个对象数据结构的数组，以对象的序号为下标就可得到具体对象的起始地址。
- 每个 slab 上还有个对象链接数组，用来实现一个空闲对象链。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- 同时，每个 slab 的描述结构中都有一个字段，表明该 slab 上的第一个空闲对象。这个字段与对象链接数组结合在一起形成了一条空闲对象链。
- 在 slab 描述结构中还有一个已经分配使用的对象的计数器，当将一个空闲的对象分配使用时，就将 slab 控制结构中的计数器加 1，并将该对象从空闲队列中脱链。
- 当释放一个对象时，只需要调整链接数组中的相应元素以及 slab 描述结构中的计数器，并且根据该 slab 的使用情况而调整其在 slab 队列中的位置（例如，如果 slab 上所有的对象都已分配使用，就要将该 slab 从第二截转移到第一截去）。
- 每个 slab 的头部有一小小的区域是不使用的，称为“着色区”（coloring area）。着色区的大小使 slab 中的每个对象的起始地址都按高速缓存中的“缓冲行”（cache line）大小（80386 的一级高速缓存中缓存行大小为 16 个字节，Pentium 为 32 个字节）对齐。每个 slab 都是从一个页面边界开始的，所以本来就自然按高速缓存的缓冲行对齐，而着色区的设置只是将第一个对象的起始地址往后推到另一个与缓冲行对齐的边界。同一个对象的缓冲队列中的各个 slab 的着色区的大小尽可能地安排成不同的大小，使得不同 slab 上同一相对位置的对象的起始地址在高速缓存中互相错开，这样可以改善高速缓存的效率。
- 每个 slab 上最后一个对象以后也有一个小小的废料区是不用的，这是对着色区大小的补偿，其大小取决于着色区的大小以及 slab 与其每个对象的相对大小。但该区域与着色区的总和对于同一种对象的各个 slab 是个常数。
- 每个对象的大小基本上是所需数据结构的大小。只有当数据结构的大小不与高速缓存中的缓冲行对齐时，才增加若干字节使其对齐。所以，一个 slab 上的所有对象的起始地址都必然是按高速缓存中的缓冲行对齐的。

下面就是 slab 描述结构 slab_t 的定义，在 mm/slab.c 中：

==================== mm/slab.c 138 152 ====================

```
/*
 * slab_t
 *
 * Manages the objs in a slab. Placed either at the beginning of mem allocated
 * for a slab, or allocated from an general cache.
 * Slabs are chained into one ordered list: fully used, partial, then fully
 * free slabs.
 */
typedef struct slab_s {
	struct list_head	list;
	unsigned long		colouroff;
	void			*s_mem;		/* including colour offset */
	unsigned int		inuse;		/* num of objs active in slab */
	kmem_bufctl_t		free;
} slab_t;
```

这里的队列头 list 用来将一块 slab 链入一个专用缓冲区队列，colouroff 为本 slab 上着色区的大小，指针 s_mem 指向对象区的起点，inuse 是已分配对象的计数器。最后，free 的值指明了空闲对象链中的

第一个对象，其实是个整数：

==================== mm/slab.c 110 131 ====================

```
/*
 * kmem_bufctl_t:
 *
 * Bufctl's are used for linking objs within a slab
 * linked offsets.
 *
 * This implementaion relies on "struct page" for locating the cache &
 * slab an object belongs to.
 * This allows the bufctl structure to be small (one int), but limits
 * the number of objects a slab (not a cache) can contain when off-slab
 * bufctls are used. The limit is the size of the largest general cache
 * that does not use off-slab slabs.
 * For 32bit archs with 4 kB pages, is this 56.
 * This is not serious, as it is only for large objects, when it is unwise
 * to have too many per slab.
 * Note: This limit can be raised by introducing a general cache whose size
 * is less than 512 (PAGE_SIZE<<3), but greater than 256.
 */

#define BUFCTL_END 0xffffFFFF
#define  SLAB_LIMIT 0xffffFFFE
typedef unsigned int kmem_bufctl_t;
```

在空闲对象链接数组中，链内每一个对象所对应元素的值为下一个对象的序号，最后一个对象所对应元素的值为 BUFCTL_END。

为每种对象建立的 slab 队列都有个队列头，其控制结构为 kmem_cache_t。该数据结构中除用来维持 slab 队列的各种指引外，还记录了适用于队列中每个 slab 的各种参数，以及两个函数指针：一个是对象的构造函数（constructor），另一个是拆除函数（destructor）。有趣的是，像其它数据结构一样，每种对象的 slab 队列头也是在 slab 上。系统中有个总的 slab 队列，其对象是各个其它对象的 slab 队列头，其队列头则也是一个 kmem_cache_t 结构，称为 cache_cache。

这样，就形成了一种层次式的树形结构；

- 总根 cache_cache 是一个 kmem_cache_t 结构，用来维持第一层 slab 队列，这些 slab 上的对象都是 kmem_cache_t 数据结构。
- 每个第一层 slab 上的每个对象，即 kmem_cache_t 数据结构都是队列头，用来维持一个第二层 slab 队列。
- 第二层 slab 队列基本上都是为某种对象，即数据结构专用的。
- 每个第二层 slab 上都维持着一个空闲对象队列。

总体的组织如下页图 2.9 所示。

从图 2.9 中可以看出，最高的层次是 slab 队列 cache_cache，队列中的每个 slab 载有若干个 kmem_cache_t 数据结构。而每个这样的数据结构又是某种数据结构（例如 inode、vm_area_struct、mm_struct，乃至于 IP 网络信息包等等）缓冲区的 slab 队列的头部。这样，当要分配一个某种数据结构

的缓冲区时，就只要指明是从哪一个队列中分配，而不需要说明缓冲区的大小，并且不需要初始化了。具体的函数是：

```
void *kmem_cache_alloc(kmem_cache_t *cachep, int flags);
void *kmem_cache_free(kmem cache_t *cachep, void *objp);
```

所以，当需要分配一个具有专用 slab 队列的数据结构时，应该通过 kmem_cache_alloc()分配。例如，我们在本章中看到过的 mm_struct、vm_area_struct、file、dentry、inode 等常用的数据结构，就都有专用的 slab 队列，而应通过 kmem_cache_alloc()分配。

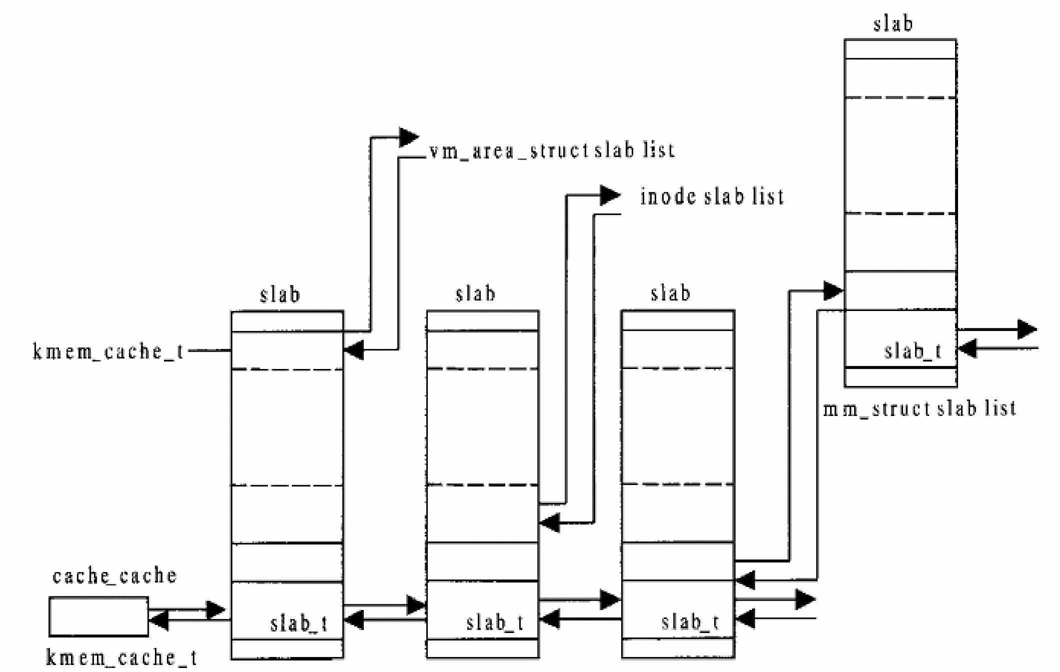


图 2.9　小对象缓冲区结构示意图

当数据结构比较大，因而不属于“小对象”时，slab 的结构略有不同。不同之处是不将 slab 的控制结构放在它所代表的 slab 上，而是将其游离出来，集中放在另外的 slab 上。由于在 slab 的控制结构 kmem_slab_t 中有一个指针指向相应 slab 上的第一个对象，所以逻辑上是一样的。其实，这就是将控制结构与控制对象相分离的一般模式。打个比方，载有“小对象”的 slab 就好像是随身携带着的户口本，而载有“大对象”的 slab 就好像是将户口本集中存放在派出所里或者是某个代理机构里。此外，当对象的大小恰好是物理页面的 1/2、1/4 或 1/8 时，将依附于每个对象的链接指针紧挨着放在一起会造成 slab 空间上的重大浪费，所以在这些特殊情况下，将链接指针也从 slab 上游离出来集中存放，以提高 slab 的空间使用率。

不过，并非内核中使用的所有数据结构都有必要拥有专用的缓冲区队列，一些不太常用、初始化开销也不大的数据结构还是可以合用一个通用的缓冲区分配机制。所以，Linux 内核中还有一种既类似于物理页面分配中采用的按大小分区，又采用 slab 方式管理的通用缓冲池，称为“ slab_cache”。slab_cache

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的结构与 cache_cache 大同小异，只不过其顶层不是一个队列而是一个结构数组（这是由于 slab_cache 相对来说比较静态），数组中的每一个元素指向一个不同的 slab 队列。这些 slab 队列的不同之处仅在于所载对象的大小。最小的是 32，然后依次为 64、128、…直至 128K（也就是 32 个页面）。从通用缓冲池中分配和释放缓冲区的函数为：

```
void *kmalloc(size_t size, int flags);
void kfree(const void *objp);
```

所以，当需要分配一个不具有专用 slab 队列的数据结构而又不必为之使用整个页面时，就应该通过 kmalloc()分配。这一般都是些细小而又不常用的数据结构，例如第 5 章中安装文件系统时使用的 vfsmount 数据结构就是这样。如果数据结构的大小接近于一个页面，则也可以干脆就通过 alloc_pages()为之分配一个页面。

顺便提一下，内核中还有一组与内存分配有关的函数 vmalloc()和 vfree()：

```
void *vmalloc(unsigned long size);
void vfree(void* addr);
```

函数 vmalloc()从内核的虚存空间（3G 以上）分配一块虚存以及相应的物理内存，类似于系统调用 brk()。不过 brk()是由进程在用户空间启动并从用户空间中分配的，而 vmalloc()则是在系统空间，也就是内核中启动，从内核空间中分配的。由 vmalloc()分配的空间不会被 kswapd 换出，因为 kswapd 只扫描各个进程的用户空间，而根本就看不到通过 vmalloc()分配的页面表项。至于通过 kmalloc()分配的数据结构，则 kswapd 只是从各个 slab 队列中寻找和收集空闲不用的 slab，并释放所占用的页面，但是不会将尚在使用中的 slab 所占据的页面换出。由于 vmalloc()与我们后面要讲的 ioremap()非常相似，这里就不讲了。

在讲解内核缓冲区的分配之前，我们先介绍缓冲区队列的建立。

2.10.1 专用缓冲区队列的建立

本来，虚存区间结构 vm_area_struct 的专用缓冲区队列是一个很好的实例，读者都已经熟悉了这个数据结构的使用。但是，到现在为止，Linux 内核中多数专用缓冲区的建立都用 NULL 作为构造函数（constructor）的指针，也就是说并没有充分利用 slab 管理机制所提供的好处（相对来说，slab 是比较新的技术），似乎不够典型。所以，我们从内核的网络驱动子系统中选择了一个例了，这是在 net/core/skbuff.c 中定义的：

==================== net/core/skbuff.c 473 487 ====================

```
void __init skb_init(void)
{
	int i;

	skbuff_head_cache = kmem_cache_create("skbuff_head_cache",
					      sizeof(struct sk_buff),
					      0,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                             SLAB_HWCACHE_ALIGN,
                             skb_headerinit, NULL);
    if (!skbuff_head_cache)
        panic("cannot create skbuff cache");

    for (i=0; i<NR_CPUS; i++)
        skb_queue_head_init(&skb_head_pool[i].list);
}
```

从代码中可以看到，skb_init 所做的事情实际上就是为网络驱动子系统建立一个 sk_buff 数据结构的专用缓冲区队列，其名称为“skbuff_head_cache”。读者可用命令“cat /proc/slabinfo”来观察这些队列的使用情况。每个缓冲区，或者说“对象”的大小是 sizeof(struct sk_buff)。调用参数 offset 为 0，表示对第一个缓冲区在 slab 中的位移并无特殊要求。但是参数 flags 为 SLAB_HWCACHE_ALIGN，表示要求与高速缓存中的缓冲行边界（16 字节或 32 字节）对齐。对象的构造函数为 skb_headerinit()，而 destructor 则为 NULL，也就是说在拆除或释放一个 slab 时无需对各个缓冲区进行特殊的处理。

函数 kmem_cache_create()所做的事情过于专门，过于冷僻，这里就不深入到其代码中去了，只是把它的内容概要介绍如下。

首先，要从 cache_cache 中分配一个 kmem_cache_t 结构，作为 sk_buff 数据结构 slab 队列的控制结构。数据结构类型 kmem_cache_t 是在 mm/slab.c 中定义的：

==================== mm/slab.c 181 237 ====================

```
struct kmem_cache_s {
/* 1) each alloc & free */
    /* full, partial first, then free */
    struct list_head   slabs;
    struct list_head   *firstnotfull;
    unsigned int       objsize;
    unsigned int       flags;   /* constant flags */
    unsigned int       num; /* # of objs per slab */
    spinlock_t         spinlock;
#ifdef CONFIG_SMP
    unsigned int       batchcount;
#endif

/* 2) slab additions /removals */
    /* order of pgs per slab (2^n) */
    unsigned int       gfporder;

    /* force GFP flags, e.g. GFP_DMA */
    unsigned int       gfpflags;

    size_t             colour;       /* cache colouring range */
    unsigned int       colour_off;   /* colour offset */
    unsigned int       colour_next;  /* cache colouring */
    kmem_cache_t       *slabp_cache;
    unsigned int       growing;
    unsigned int       dflags;       /* dynamic flags */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

	/* constructor func */
	void (*ctor)(void *, kmem_cache_t *, unsigned long);

	/* de-constructor func */
	void (*dtor)(void *, kmem_cache_t *, unsigned long);

	unsigned long		failures;

/* 3) cache creation/removal */
	char			name[CACHE_NAMELEN];
	struct list_head	next;
#ifdef CONFIG_SMP
/* 4) per-cpu data */
	cpucache_t		*cpudata[NR_CPUS];
#endif
#if STATS
	unsigned long		num_active;
	unsigned long		num_allocations;
	unsigned long		high_mark;
	unsigned long		grown;
	unsigned long		reaped;
	unsigned long		errors;
#ifdef CONFIG_SMP
	atomic_t		allochit;
	atomic_t		allocmiss;
	atomic_t		freehit;
	atomic_t		freemiss;
#endif
#endif
};
```

在 kmem_cache_s 的基础上，在 include/linux/slab.h 中又定义了 kmem_cache_t：

===================== include/linux/slab.h 12 12 =====================

```
typedef struct kmem_cache_s kmem_cache_t;
```

结构中的队列头 slabs 用来维持一个 slab 队列，指针 firstnotfull 则指向队列中第一个含有空闲对象（即缓冲区）的 slsb，也就是指向队列中的第 2 段。当这个指针指向队列头 slabs 时就表明队列中不存在含有空闲对象 slab。

结构中还有个队列头 next，则是用来在 cache_cache 中建立一个“专用缓冲区 slab 队列的队列”，也就是 slab 队列控制结构的队列。当 slab 的描述结构与对象不在同一 slab 上时，即对于大对象 slab，指针 slabp_cache 指向对方队列的控制结构。

除这些队列头和指针以外，还有一些重要的成分：objsize 是原始的数据结构（对象）的大小，在这个情景中就是 sizeof(struct sk_buff)；num 表示每个 slab 上有几个缓冲区；gfporder 则表示每个 slab 的大小，每个 slab 都是由 2^n 个页面构成的，而 gfporder 就是 n。

前面讲过，在每个 slab 的前部保留了一小块区域空着不用，那就是“着色区”（coloring area），其

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

作用是使同一 slab 队列中不同 slab 上对象区的起始地址互相错开，这样有利于改善高速缓冲的效率。所以，如果当前 slab 的颜色为 1，则下一个 slab 的颜色将是 2，使下一个 slab 中的第一个缓冲区更往后推一些。但是，不同“颜色”的数量是有限的，它取决于一块 slab 分割成若干缓冲区（对象）以及所需的其它空间以后的剩余，以及高速缓存中每个缓冲行（cache line）的大小。所以，对每个 slab 队列都要计算出它的颜色数量，这个数量就保存在 colour 中，而下一个 slab 将要使用的颜色则保存在 colour_next 中。当 colour_next 达到最大值 colour 时，就又从 0 开始，如此周而复始。着色区的大小可以根据(colour_off×colour)算得。

分配了一个 kmem_cache_t 结构以后，kmem_cache_create()就进行一系列的计算，以确定最佳的 slab 构成。包括：每个 slab 由几个页面组成，划分成多少个缓冲区（即“对象”）；slab 的控制结构 kmem_slab_t 应该在 slab 外面集中存放还是就放在每个 slab 的尾部；每个缓冲区的链接指针应该在 slab 外面集中存放还在 slab 上与相应的缓冲区紧挨着放在一起；还有“颜色”的数量等等。并根据调用参数和计算的结果设置队列头 kmem_cache_t 结构中的各个参数，包括两个函数指针 ctor 和 dtor。

最后，将队头 kmem_cache_t 结构链入 cache_cache 的 next 队列中（注意，不是它的 slab 队列中）。

函数 kmem_cache_create()只是建立了所需的专用缓冲区队列的基础设施，所形成的 slab 队列是个空队列。而具体 slab 的创建则要等需要分配缓冲区时，却发现队列中并无空闲的缓冲区可供分配时，再通过 kmem_cache_grow()来进行。

2.10.2 缓冲区的分配与释放

在建立了一种缓冲区的专用队列以后，就可以通过 kmem_cache_alloc()来分配缓冲区了。就上面建立的 skbuff_head_cache 队列来说，文件 net/core/skbuff.c 中是这样进行分配的：

==================== net/core/skbuff.c 165 166 ====================

```
struct sk_buff *alloc_skb(unsigned int size,int gfp_mask)
{
  ......
```

==================== net/core/skbuff.c 181 186 ====================

```
     skb = skb_head_from_pool();
     if (skb == NULL) {
          skb = kmem_cache_alloc(skbuff_head_cache, gfp_mask);
          if (skb == NULL)
               goto nohead;
     }
  ......
```

==================== net/core/skbuff.c 215 215 ====================

```
}
```

函数 alloc_skb()是具体设备驱动程序对 kmem_cache_alloc()的包装，在此基础上建立起自己的缓冲区管理机制，包括一个 sk_buff 数据结构的缓冲池，以加快分配数据结构的速度，并防止因具体驱动程序分配／释放缓冲区不当而引起问题。这样，就把具体的设备驱动程序与 kmem_cache_alloc()分隔开了。

要分配一个 sk_buff 数据结构，先通过 skb_head_from_pool()试试缓冲池。如果在缓冲池中得不到，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

那就要进一步通过 kmem_cache_alloc()分配，这就是我们所关心的。其代码在 mm/slab.c 中：

==================== mm/slab.c 1506 1509 ====================

[alloc_skb()>kmem_cache_alloc()]

```
void * kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
     return __kmem_cache_alloc(cachep, flags);
}
```

==================== mm/slab.c 1291 1299 ====================

[alloc_skb()>kmem_cache_alloc()>__kmem_cache_alloc()]

```
static inline void * __kmem_cache_alloc (kmem_cache_t *cachep, int flags)
{
     unsigned long save_flags;
     void* objp;

     kmem_cache_alloc_head(cachep, flags);
try_again:
     local_irq_save(save_flags);
#ifdef CONFIG_SMP
   ......
```

==================== mm/slab.c 1319 1325 ====================

```
#else
     objp = kmem_cache_alloc_one(cachep);
#endif
     local_irq_restore(save_flags);
     return objp;
alloc_new_slab:
#ifdef CONFIG_SMP
   ......
```

==================== mm/slab.c 1328 1336 ====================

```
#endif
     local_irq_restore(save_flags);
     if (kmem_cache_grow(cachep, flags))
          /* Someone may have stolen our objs.  Doesn't matter, we'll
           * just come back here again.
           */
          goto try_again;
     return NULL;
}
```

首先，alloc_skb()中的指针 skbuff_head_cache 是个全局量，指向相应的 slab 队列（这里是 sk_buff 结构的 slab 队列）的队列头，因而这里的参数 cachep 也指向这个队列。

程序中的 kmem_cache_alloc_head()是为调试而设的，在实际运行的系统中是空函数。我们在这里也不关心多处理器 SMP 结构，所以这里关键性的操作就是 kmem_cache_alloc_one()，这是一个宏操作，其定义为：

==================== mm/slab.c 1246 1263 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1246 /*
1247  * Returns a ptr to an obj in the given cache.
1248  * caller must guarantee synchronization
1249  * #define for the goto optimization 8-)
1250  */
1251 #define kmem_cache_alloc_one(cachep)                \
1252 ({                                  \
1253      slab_t    *slabp;                    \
1254                                      \
1255      /* Get slab alloc is to come from. */           \
1256      {                               \
1257           struct list_head* p = cachep->firstnotfull;    \
1258           if (p == &cachep->slabs)              \
1259                goto alloc_new_slab;             \
1260           slabp = list_entry(p,slab_t, list);   \
1261      }                               \
1262      kmem_cache_alloc_one_tail(cachep, slabp);       \
1263 })
```

上面__kmem_cache_alloc()的代码一定要和这个宏定义结合起来看才能明白。从定义中可以看到，第一步是通过 slab 队列头中的指针 firstnotfull，找到该队列中第一个含有空闲对象的 slab。如果这个针指向 slab 队列的链头（是链中的第一个 slab），那就表示队列中已经没有含有空闲对象的 slab，所以就转到__kmem_cache_alloc()中的标号 alloc_new_slab 处（1324 行），进一步扩充该 slab 队列。

如果找到了含有空闲对象的 slab，就调用 kmem_cache_alloc_tail()分配一个空闲对象并返回其指针：

===================== mm/slab.c 1211 1228 =====================

[alloc_skb()>kmem_cache_alloc()>__kmem_cache_alloc()>kmem_cache_alloc_one_tail()]

```
1211 static inline void * kmem_cache_alloc_one_tail (kmem_cache_t *cachep,
1212                               slab_t *slabp)
1213 {
1214      void *objp;
1215
1216      STATS_INC_ALLOCED(cachep);
1217      STATS_INC_ACTIVE(cachep);
1218      STATS_SET_HIGH(cachep);
1219
1220      /* get obj pointer */
1221      slabp->inuse++;
1222      objp = slabp->s_mem + slabp->free*cachep->objsize;
1223      slabp->free=slab_bufctl(slabp)[slabp->free];
1224
1225      if (slabp->free == BUFCTL_END)
1226           /* slab now full: move to next slab for next alloc */
1227           cachep->firstnotfull = slabp->list.next;
1228 #if DEBUG
   ......
```

==================== mm/slab.c 1242 1244 ===================

```
1242 #endif
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1243      return objp;
1244 }
```

如前所述，数据结构 slab_t 中的 free 记录着下一次可以分配的空闲对象的序号，而 s_mem 则指向 slab 中的对象区，所以根据这些数据和本专用队列的对象大小，就可以计算出该空闲对象的起始地址。然后，就通过宏操作 slab_bufctl()改变字段 free 的值，使它指明下一个空闲对象的序号。

==================== mm/slab.c 154 155 ====================

```
154  #define slab_bufctl(slabp) \
155       ((kmem_bufctl_t *)(((slab_t*)slabp)+1))
```

这个宏操作返回一个 kmem_bufctl_t 数组的地址，这个数组就在 slab 中数据结构 slab_t 的上方，紧挨着数据结构 slab_t。该数组以当前对象的序号为下标，而数组元素的值则表明下一个空闲对象的序号。改变了 slab_t 中 free 字段的值，就隐含着当前对象已被分配。

如果达到了 slab 的末尾 BUFCTL_END，就要调整该 slab 队列的指针 firstnotfull，使它指向队列中的下一个 slab。

不过，我们假定 slab 队列中已经不存在含有空闲对象的 slab，所以要转到前面代码中的标号 alloc_new_slab 处，通过 kmem_cache_grow()来分配一块新的 slab，使缓冲区的队列“生长”起来。函数 kmem_cache_grow()的代码也在 mm/slab.c 中：

==================== mm/slab.c 1066 1168 ====================

[alloc_skb()>kmem_cache_alloc()>__kmem_cache_alloc()>kmem_cache_grow()]

```
1066 /*
1067  * Grow (by 1) the number of slabs within a cache.  This is called by
1068  * kmem_cache_alloc() when there are no active objs left in a cache.
1069  */
1070 static int kmem_cache_grow (kmem_cache_t * cachep, int flags)
1071 {
1072      slab_t    *slabp;
1073      struct page    *page;
1074      void      *objp;
1075      size_t          offset;
1076      unsigned int    i, local_flags;
1077      unsigned long  ctor_flags;
1078      unsigned long  save_flags;
1079 
1080      /* Be lazy and only check for valid flags here,
1081       * keeping it out of the critical path in kmem_cache_alloc().
1082       */
1083      if (flags & ~(SLAB_DMA|SLAB_LEVEL_MASK|SLAB_NO_GROW))
1084           BUG();
1085      if (flags & SLAB_NO_GROW)
1086           return 0;
1087 
1088      /*
1089       * The test for missing atomic flag is performed here, rather than
1090       * the more obvious place, simply to reduce the critical path length
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 * in kmem_cache_alloc(). If a caller is seriously mis-behaving they
	 * will eventually be caught here (where it matters).
	 */
	if (in_interrupt() && (flags & SLAB_LEVEL_MASK) != SLAB_ATOMIC)
		BUG();

	ctor_flags = SLAB_CTOR_CONSTRUCTOR;
	local_flags = (flags & SLAB_LEVEL_MASK);
	if (local_flags == SLAB_ATOMIC)
		/*
		 * Not allowed to sleep.  Need to tell a constructor about
		 * this - it might need to know...
		 */
		ctor_flags |= SLAB_CTOR_ATOMIC;

	/* About to mess with non-constant members - lock. */
	spin_lock_irqsave(&cachep->spinlock, save_flags);

	/* Get colour for the slab, and cal the next value. */
	offset = cachep->colour_next;
	cachep->colour_next++;
	if (cachep->colour_next >= cachep->colour)
		cachep->colour_next = 0;
	offset *= cachep->colour_off;
	cachep->dflags |= DFLGS_GROWN;

	cachep->growing++;
	spin_unlock_irqrestore(&cachep->spinlock, save_flags);

	/* A series of memory allocations for a new slab.
	 * Neither the cache-chain semaphore, or cache-lock, are
	 * held, but the incrementing c_growing prevents this
	 * cache from being reaped or shrunk.
	 * Note: The cache could be selected in for reaping in
	 * kmem_cache_reap(), but when the final test is made the
	 * growing value will be seen.
	 */

	/* Get mem for the objs. */
	if (!(objp = kmem_getpages(cachep, flags)))
		goto failed;

	/* Get slab management. */
	if (!(slabp = kmem_cache_slabmgmt(cachep, objp, offset, local_flags)))
		goto opps1;

	/* Nasty!!!!!! I hope this is OK. */
	i = 1 << cachep->gfporder;
	page = virt_to_page(objp);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    do {
        SET_PAGE_CACHE(page, cachep);
        SET_PAGE_SLAB(page, slabp);
        PageSetSlab(page);
        page++;
    } while (--i);

    kmem_cache_init_objs(cachep, slabp, ctor_flags);

    spin_lock_irqsave(&cachep->spinlock, save_flags);
    cachep->growing--;

    /* Make slab active. */
    list_add_tail(&slabp->list,&cachep->slabs);
    if (cachep->firstnotfull == &cachep->slabs)
        cachep->firstnotfull = &slabp->list;
    STATS_INC_GROWN(cachep);
    cachep->failures = 0;

    spin_unlock_irqrestore(&cachep->spinlock, save_flags);
    return 1;
opps1:
    kmem_freepages(cachep, objp);
failed:
    spin_lock_irqsave(&cachep->spinlock, save_flags);
    cachep->growing--;
    spin_unlock_irqrestore(&cachep->spinlock, save_flags);
    return 0;
}
```

函数 kmem_cache_grow()根据队列头中的参数 gfporder 分配若干连续的物理内存页面，并将这些页面构造成 slab，链入给定的 slab 队列。对参数进行了一些检查以后，就计算出下一块 slab 应有的着色区大小。然后，通过 kmem_getpages()分配用于具体对象缓冲区的页面，这个函数最终调用 alloc_pages()分配空闲页面。分配了用于对象本身的内存页面后，还要通过 kmem_cache_slabmgmt()建立起 slab 的管理信息。其代码在 mm/slab.c 中：

==================== mm/slab.c 996 1021 ====================

[alloc_skb()>kmem_cache_alloc()>__kmem_cache_alloc()>kmem_cache_grow()>kmem_cache_slabmgmt()]

```
/* Get the memory for a slab management obj. */
static inline slab_t * kmem_cache_slabmgmt (kmem_cache_t *cachep,
              void *objp, int colour_off, int local_flags)
{
    slab_t *slabp;

    if (OFF_SLAB(cachep)) {
        /* Slab management obj is off-slab. */
        slabp = kmem_cache_alloc(cachep->slabp_cache, local_flags);
        if (!slabp)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1006                    return NULL;
1007         } else {
1008                /* FIXME: change to
1009                    slabp = objp
1010                 * if you enable OPTIMIZE
1011                 */
1012                slabp = objp+colour_off;
1013                colour_off += L1_CACHE_ALIGN(cachep->num *
1014                        sizeof(kmem_bufctl_t) + sizeof(slab_t));
1015         }
1016         slabp->inuse = 0;
1017         slabp->colouroff = colour_off;
1018         slabp->s_mem = objp+colour_off;
1019 
1020         return slabp;
1021 }
```

如前所述，小对象的 slab 控制结构 slab_t 与对象本身共存于同一 slab 上，而大对象的控制结构则游离于 slab 之外。但是，大对象的控制结构也是 slab_t，存在于为这种数据结构专设的 slab 上，也有其专用的 slab 队列。所以，如果是大对象就通过 kmem_cache_alloc()分配一个 slab_t，否则就用小对象 slab 低端的一部分空间用作其控制结构，不过在此之前要空出一小块着色区。注意这里 1012 行和 1017 行所引用的 colour_off 不是同一个数值，这个变量的值已在 1013 行作了调整，在原来的数值上增加了对象链接数组的大小以及控制结构 slab_t 的大小。所以，slabp->s_mem 总是指向 slab_t 上对象区的起点。

对分配用于 slab 的每个页面的 page 数据结构，要通过宏操作 SET_PAGE_CACHE 和 SET_PAGE_SLAB，设置其链接指针 prev 和 next，使它们分别指向所属的 slab 和 slab 队列。同时，还要把 page 结构中的 PG_slab 标志位设成 1，以表明该页面的用途。

最后，通过 kmem_cache_init_objs()进行 slab 的初始化：

==================== mm/slab.c 1023 1030 ====================

[alloc_skb()>kmem_cache_alloc()>__kmem_cache_alloc()>kmem_cache_grow()>kmem_cache_init_objs()]

```
static inline void kmem_cache_init_objs (kmem_cache_t * cachep,
            slab_t * slabp, unsigned long ctor_flags)
{
    int i;

    for (i = 0; i < cachep->num; i++) {
        void* objp = slabp->s_mem+cachep->objsize*i;
#if DEBUG
  ......
```

==================== mm/slab.c 1037 1046 ====================

```
#endif

        /*
         * Constructors are not allowed to allocate memory from
         * the same cache which they are a constructor for.
```

```
1042           * Otherwise, deadlock. They must also be threaded.
1043           */
1044          if (cachep->ctor)
1045              cachep->ctor(objp, cachep, ctor_flags);
1046 #if DEBUG
   ......
```

==================== mm/slab.c 1059 1064 ====================

```
1059 #endif
1060          slab_bufctl(slabp)[i] = i+1;
1061      }
1062      slab_bufctl(slabp)[i-1] = BUFCTL_END;
1063      slabp->free = 0;
1064 }
```

这里的初始化包括了对具体对象构造函数的调用。对于 sk_buff 数据结构，这个函数就是 skb_headerinit()。此外，代码中的 1060 行是对链接数组中各个元素的初始化。

缓冲区队列“成长”了一些以后，就一定有空闲缓冲区可供分配了。所以转回标号 try_again 处再试一遍（见__kmem_cache_alloc()中的 1334 行）。

这样，就构成了一个多层次的缓冲区分配机制。位于最高层的是缓冲区的分配，在我们这个情景中就是 alloc_skb()，具体则是先通过 skb_head_from_pool()，从缓冲池，即已经分配的 slab 块中分配。如果失败的话，就往下跑一层从 slab 队列中通过 kmem_cache_alloc()分配。要是 slab 队列中已经没有载有空闲缓冲区的 slab，那就再往下跑一层，通过 kmem_cache_grow()，分配若干页面而构造出一个 slab 块。

那么，缓冲区队列是否单调地成长而不缩小呢？我们在以前提到过，kswapd 定时地调用 kmem_cache_reap()来“收割”。也就是说，依次检查若干专用缓冲区 slab 队列，看看是否有完全空闲的 slab 存在。有的话就将这些 slab 占用的内存页面释放。

再来看专用缓冲区的释放，这是由 kmem_cache_free()完成的。其代码在 mm/slab.c 中：

==================== mm/slab.c 1554 1557 ====================

```
1554 void kmem_cache_free (kmem_cache_t *cachep, void *objp)
1555 {
1556      unsigned long flags;
1557 #if DEBUG
   ......
```

==================== mm/slab.c 1561 1566 ====================

```
1561 #endif
1562
1563      local_irq_save(flags);
1564      __kmem_cache_free(cachep, objp);
1565      local_irq_restore(flags);
1566 }
```

显然，操作的主体是__kmem_cache_free()，这里只是在操作期间把中断暂时关闭。

==================== mm/slab.c 1466 1472 ================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

[kmem_cache_free()>__kmem_cache_free()]

```
1466 /*
1467  * __kmem_cache_free
1468  * called with disabled ints
1469  */
1470 static inline void __kmem_cache_free (kmem_cache_t *cachep, void* objp)
1471 {
1472 #ifdef CONFIG_SMP
    ......
```

==================== mm/slab.c 1493 1496 ====================

```
1493 #else
1494      kmem_cache_free_one(cachep, objp);
1495 #endif
1496 }
```

我们在这里不关心多处理器 SMP 结构，而函数 kmem_cache_free_one()的代码也在同一文件中：

==================== mm/slab.c 1367 1380 ====================

[kmem_cache_free()>__kmem_cache_free()>kmem_cache_free_one()]

```
1367 static inline void kmem_cache_free_one(kmem_cache_t *cachep, void *objp)
1368 {
1369      slab_t* slabp;
1370
1371      CHECK_PAGE(virt_to_page(objp));
1372      /* reduces memory footprint
1373       *
1374      if (OPTIMIZE(cachep))
1375           slabp = (void*)((unsigned long)objp&(~(PAGE_SIZE-1)));
1376       else
1377       */
1378      slabp = GET_PAGE_SLAB(virt_to_page(objp));
1379
1380 #if DEBUG
    ......
```

==================== mm/slab.c 1402 1448 ====================

```
1402 #endif
1403      {
1404           unsigned int objnr = (objp-slabp->s_mem)/cachep->objsize;
1405
1406           slab_bufctl(slabp)[objnr] = slabp->free;
1407           slabp->free = objnr;
1408      }
1409      STATS_DEC_ACTIVE(cachep);
1410
1411      /* fixup slab chain */
1412      if (slabp->inuse-- == cachep->num)
1413           goto moveslab_partial;
1414      if (!slabp->inuse)
1415           goto moveslab_free;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1416     return;
1417
1418 moveslab_partial:
1419     /* was full.
1420      * Even if the page is now empty, we can set c_firstnotfull to
1421      * slabp: there are no partial slabs in this case
1422      */
1423     {
1424         struct list_head *t = cachep->firstnotfull;
1425
1426         cachep->firstnotfull = &slabp->list;
1427         if (slabp->list.next == t)
1428             return;
1429         list_del(&slabp->list);
1430         list_add_tail(&slabp->list, t);
1431         return;
1432     }
1433 moveslab_free:
1434     /*
1435      * was partial, now empty.
1436      * c_firstnotfull might point to slabp
1437      * FIXME: optimize
1438      */
1439     {
1440         struct list_head *t = cachep->firstnotfull->prev;
1441
1442         list_del(&slabp->list);
1443         list_add_tail(&slabp->list, &cachep->slabs);
1444         if (cachep->firstnotfull == &slabp->list)
1445             cachep->firstnotfull = t->next;
1446         return;
1447     }
1448 }
```

代码中的 CHECK_PAGE 只用于程序调试，在实际运行的系统中为空语句。根据待释放对象的地址可以算出其所在的页面。进一步，如前所述（见 kmem_cache_grow()中的 1142 行），页面的 page 结构中链头 list 内，原用于队列链接的指针 prev，指向页面所属的 slab，所以通过宏操作 GET_PAGE_SLAB 就可以得到这个 slab 的指针。找到了对象所在的 slab，就可以通过其链接数组释放给定对象了（见 1404～1407 行）。同时，还要递减所属 slab 队列控制结构中非空闲对象的计数。递减以后有三种可能：

- 原来 slab 没有空闲对象，而现在有了，所以要转到 moveslab_partial 处，把 slab 从队列中原来的位置移到队列的第二截，即由指针 firstnotfull 所指的地方。
- 原来 slab 上就有空闲对象，而现在所有对象都空闲了，所以要转到 moveslab_free 处，把 slab 从队列中原来的位置移到队列的第三截，即队列的末尾。
- 原来 slab 上就有空闲对象，现在只不过是多了一个，但也并没有全部空闲，所以不需要任何改动。

可见，分配和释放专用缓冲区的开销都是很小的。这里还要指出，缓冲区的释放并不导致 slab 的

PDF-XCHANGE Click to buy NOW! www.docu-track.com

释放，空闲 slab 的释放是由 kswapd 等内核线程周期地调用 kmem_cache_reap()完成的。

看完了专用缓冲区的分配和释放，再看看通用缓冲区的分配。前面讲过，除各种专用的缓冲区队列外，内核中还有一个通用的缓冲池 cache_sizes，里面根据缓冲区的大小而分成若干队列。用了通用缓冲区分配的函数 kmalloc()是在 mm/slab.c 中定义的：

==================== mm/slab.c 1511 1544 ====================

```
/**
 * kmalloc - allocate memory
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate.
 *
 * kmalloc is the normal method of allocating memory
 * in the kernel.  The @flags argument may be one of:
 *
 * %GFP_BUFFER - XXX
 *
 * %GFP_ATOMIC - allocation will not sleep.  Use inside interrupt handlers.
 *
 * %GFP_USER - allocate memory on behalf of user.  May sleep.
 *
 * %GFP_KERNEL - allocate normal kernel ram.  May sleep.
 *
 * %GFP_NFS - has a slightly lower probability of sleeping than %GFP_KERNEL.
 * Don't use unless you're in the NFS code.
 *
 * %GFP_KSWAPD - Don't use unless you're modifying kswapd.
 */
void * kmalloc (size_t size, int flags)
{
	cache_sizes_t *csizep = cache_sizes;

	for (; csizep->cs_size; csizep++) {
		if (size > csizep->cs_size)
			continue;
		return __kmem_cache_alloc(flags & GFP_DMA ?
			 csizep->cs_dmacachep : csizep->cs_cachep, flags);
	}
	BUG(); // too big size
	return NULL;
}
```

这里通过一个 for 循环，在 cache_sizes 结构数组中由小到大扫描，找到第一个能满足缓冲区大小要求的队列，然后就调用函数__kmem_cache_alloc()从该队列中分配一个缓冲区。而 kmem_cache_alloc()的作用我们在前面已经简要地介绍过了。

最后，我们来看看空闲 slab 的“收割”，即对构成空闲 slab 的页面的回收。以前我们看到过，内核线程 kswapd 在周期性的运行中会调用 kmem_cache_reap()回收这些页面。这个函数的代码在 mm/slab.c

PDF-XCHANGE Click to buy NOW! www.docu-track.com

中：

==================== mm/slab.c 1701 1742 ====================

```
/**
 * kmem_cache_reap - Reclaim memory from caches.
 * @gfp_mask: the type of memory required.
 *
 * Called from try_to_free_page().
 */
void kmem_cache_reap (int gfp_mask)
{
	slab_t *slabp;
	kmem_cache_t *searchp;
	kmem_cache_t *best_cachep;
	unsigned int best_pages;
	unsigned int best_len;
	unsigned int scan;

	if (gfp_mask & __GFP_WAIT)
		down(&cache_chain_sem);
	else
		if (down_trylock(&cache_chain_sem))
			return;

	scan = REAP_SCANLEN;
	best_len = 0;
	best_pages = 0;
	best_cachep = NULL;
	searchp = clock_searchp;
	do {
		unsigned int pages;
		struct list_head* p;
		unsigned int full_free;

		/* It's safe to test this without holding the cache-lock. */
		if (searchp->flags & SLAB_NO_REAP)
			goto next;
		spin_lock_irq(&searchp->spinlock);
		if (searchp->growing)
			goto next_unlock;
		if (searchp->dflags & DFLGS_GROWN) {
			searchp->dflags &= ~DFLGS_GROWN;
			goto next_unlock;
		}
#ifdef CONFIG_SMP
```

......

==================== mm/slab.c 1750 1825 ====================

```
#endif

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                full_free = 0;
                p = searchp->slabs.prev;
                while (p != &searchp->slabs) {
                        slabp = list_entry(p, slab_t, list);
                        if (slabp->inuse)
                                break;
                        full_free++;
                        p = p->prev;
                }

                /*
                 * Try to avoid slabs with constructors and/or
                 * more than one page per slab (as it can be difficult
                 * to get high orders from gfp()).
                 */
                pages = full_free * (1<<searchp->gfporder);
                if (searchp->ctor)
                        pages = (pages*4+1)/5;
                if (searchp->gfporder)
                        pages = (pages*4+1)/5;
                if (pages > best_pages) {
                        best_cachep = searchp;
                        best_len = full_free;
                        best_pages = pages;
                        if (full_free >= REAP_PERFECT) {
                                clock_searchp = list_entry(searchp->next.next,
                                                        kmem_cache_t,next);
                                goto perfect;
                        }
                }
next_unlock:
                spin_unlock_irq(&searchp->spinlock);
next:
                searchp = list_entry(searchp->next.next,kmem_cache_t,next);
        } while (--scan && searchp != clock_searchp);

        clock_searchp = searchp;

        if (!best_cachep)
                /* couldn't find anything to reap */
                goto out;

        spin_lock_irq(&best_cachep->spinlock);
perfect:
        /* free only 80% of the free slabs */
        best_len = (best_len*4 + 1)/5;
        for (scan = 0; scan < best_len; scan++) {
                struct list_head *p;

```

```
        if (best_cachep->growing)
            break;
        p = best_cachep->slabs.prev;
        if (p == &best_cachep->slabs)
            break;
        slabp = list_entry(p,slab_t,list);
        if (slabp->inuse)
            break;
        list_del(&slabp->list);
        if (best_cachep->firstnotfull == &slabp->list)
            best_cachep->firstnotfull = &best_cachep->slabs;
        STATS_INC_REAPED(best_cachep);

        /* Safe to drop the lock. The slab is no longer linked to the
         * cache.
         */
        spin_unlock_irq(&best_cachep->spinlock);
        kmem_slab_destroy(best_cachep, slabp);
        spin_lock_irq(&best_cachep->spinlock);
    }
    spin_unlock_irq(&best_cachep->spinlock);
out:
    up(&cache_chain_sem);
    return;
}
```

这个函数扫描 slab 队列的队列 cache_cache，从中发现可供“收割”的 slab 队列。不过，并不是每次都扫描整个 cache_cache，而只是扫描其中的一部分 slab 队列，所以需要有个全局量来记录下一次扫描的起点，这就是 clock_searchp：

==================== mm/slab.c 360 361 ====================

```
360 /* Place maintainer for reaping. */
361 static kmem_cache_t *clock_searchp = &cache_cache;
```

找到了可以“收割”的 slab 队列，也不是把它所有空闲的 slab 都全部回收，而是回收其中的大约 80%。对于要回收的 slab，调用 kmem_slab_destroy()释放其各个页面，我们把这个函数留给读者自己阅读。

==================== mm/slab.c 540 554 ====================

[kmem_cache_reap()>kmem_slab_destroy()]

```
540 /* Destroy all the objs in a slab, and release the mem back to the system.
541  * Before calling the slab must have been unlinked from the cache.
542  * The cache-lock is not held/needed.
543  */
544 static void kmem_slab_destroy (kmem_cache_t *cachep, slab_t *slabp)
545 {
546     if (cachep->dtor
547 #if DEBUG
548         || cachep->flags & (SLAB_POISON | SLAB_RED_ZONE)
```

```
#endif
	) {
		int i;
		for (i = 0; i < cachep->num; i++) {
			void* objp = slabp->s_mem+cachep->objsize*i;
#if DEBUG
  ......
```

==================== mm/slab.c 563 566 ====================

```
#endif
			if (cachep->dtor)
				(cachep->dtor)(objp, cachep, 0);
#if DEBUG
  ......
```

==================== mm/slab.c 573 580 ====================

```
#endif
		}
	}

	kmem_freepages(cachep, slabp->s_mem-slabp->colouroff);
	if (OFF_SLAB(cachep))
		kmem_cache_free(cachep->slabp_cache, slabp);
}
```

2.11 外部设备存储空间的地址映射

任何系统都免不了要有输入 / 输出，所以对外部设备的访问是 CPU 设计中的一个重要问题。一般来说，对外部设备的访问有两种不同的形式，一种叫内存映射式（memory mapped），另一种叫 I/O 映射式（I/O mapped）。在采用内存映射方式的 CPU 中，外部设备的存储单元，如控制寄存器、状态寄存器、数据寄存器等等，去作为内存的一部分出现在系统中的。CPU 可以像访问一个内存单元一样地访问外部设备的存储单元，所以不需要专门设立用于外设 I/O 的指令。从前的 PDP-11、后来的 M68K、Power PC 等 CPU 都采用这种方式。而在采用 I/O 映射方式的系统中则不同，外部设备的存储单元与内存分属两个不同的体系。访问内存的指令不能用来访问外部设备的存储单元，所以在 X86 CPU 中设立了专门的 IN 和 OUT 指令，但是用于 I/O 指令的“地址主间”相对来说是很小的。事实上，现在 X86 的 I/O 地址空间已经非常拥挤。

但是，随着计算机技术的发展，人们发现单纯的 I/O 映射方式是不能满足要求的。此种方式只适合于早期的计算机技术，那时候一个外设通常都只有几个寄存器，通过这几个寄存器就可以完成对外设的所有操作了。而现在的情况却大不一样。例如，在 PC 机上可以插上一块图像卡，带有 2MB 的存储器，甚至还可能带有一块 ROM，里面装有可执行代码。自从 PCI 总线出现以后，这个问题就更突出了。所以，不管 CPU 的设计采用 I/O 映射或是存储器映射，都必须要有将外设卡上的存储器映射到内存空间，实际上是虚存空间的手段。在 Linux 内核中，这样的映射是通过函数 ioremap()来建立的。

对于内存页面的管理，通常我们都是先在虚存空间分配一个虚存区间，然后为此区间分配相应的物理内存页面并建立起映射。而且这样的映射也并不是一次就建立完毕，可以在访问这些虚存页面引

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

起页面异常时逐步地建立。但是，ioremap()则不同， 首先，我们先有一个物理存储区间，其地址就是外设卡上的存储器出现在总线上的地址。这地址未必就是这些存储单元在外设卡上局部的物理地址，而是在总线上由 CPU 所“看到”的地址，这中间很可能已经经历了一次地址映射，但这种映射对于 CPU 来说是透明的。所以有时把这种地址称为“总线地址”。举例来说，如果有一块“智能图形卡”，卡上有个微处理器。对于卡上的微处理器来说，卡上的存储器是从地址 0 开始的，这就是卡上局部的物理地址。但是将这块图形卡插到 PC 的一个 PCI 总线插槽上时，由 PC 的 CPU 所看到的这片物理存储区间的地址可能是从 0x0000 f000 0000 0000 开始的，这中间已经有了一次映射。可是，从系统（PC）的 CPU 的角度来说，它只知道这片物理存储区间是从 0x0000 f000 0000 0000 开始的，这就是该区间的物理地址，或者说“总线地址”。在 Linux 系统中，CPU 不能按物理地址来访问存储中间，而必须使用虚拟地址，所以必需“反向”地从物理地址出发找到一片虚存空间并建立起映射。其次，这样的需求只发生于对外部设备的操作，而这是内核的事，所以相应的虚存区间是在系统空间（3GB 以上)。在以前的 Linux 内核版本中，这个函数称为 vremap()，后来改成了 ioremap()，也突出地反映了这一点。还有。这样的页面当然不服从动态的物理内存页面分配，也不服从 kswapd 的换出。

先看 ioremp()，这是一个 inline 函数，定义于 include/asm-i386/io.h：

==================== include/asm-i386/io.h 140 143 ====================

```
extern inline void * ioremap (unsigned long offset, unsigned long size)
{
        return __ioremap(offset, size, 0);
}
```

实际的操作由__ioremap()完成，是在 arch/i386/mm/ioremap.c 中定义的：

==================== arch/i386/mm/ioremap.c 92 152 ====================

[ioremap()>__ioremap()]

```
/*
 * Remap an arbitrary physical address space into the kernel virtual
 * address space. Needed when the kernel wants to access high addresses
 * directly.
 *
 * NOTE! We need to allow non-page-aligned mappings too: we will obviously
 * have to convert them into an offset in a page-aligned mapping, but the
 * caller shouldn't need to know that small detail.
 */
void * __ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags)
{
        void * addr;
        struct vm_struct * area;
        unsigned long offset, last_addr;

        /* Don't allow wraparound or zero size */
        last_addr = phys_addr + size - 1;
        if (!size || last_addr < phys_addr)
                return NULL;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
112     /*
113      * Don't remap the low PCI/ISA area, it's always mapped..
114      */
115     if (phys_addr >= 0xA0000 && last_addr < 0x100000)
116         return phys_to_virt(phys_addr);
117 
118     /*
119      * Don't allow anybody to remap normal RAM that we're using..
120      */
121     if (phys_addr < virt_to_phys(high_memory)) {
122         char *t_addr, *t_end;
123         struct page *page;
124 
125         t_addr = __va(phys_addr);
126         t_end = t_addr + (size - 1);
127 
128         for(page = virt_to_page(t_addr); page <= virt_to_page(t_end); page++)
129             if(!PageReserved(page))
130                 return NULL;
131     }
132 
133     /*
134      * Mappings have to be page-aligned
135      */
136     offset = phys_addr & ~PAGE_MASK;
137     phys_addr &= PAGE_MASK;
138     size = PAGE_ALIGN(last_addr) - phys_addr;
139 
140     /*
141      * Ok, go for it..
142      */
143     area = get_vm_area(size, VM_IOREMAP);
144     if (!area)
145         return NULL;
146     addr = area->addr;
147     if (remap_area_pages(VMALLOC_VMADDR(addr), phys_addr, size, flags)) {
148         vfree(addr);
149         return NULL;
150     }
151     return (void *) (offset + (char *)addr);
152 }
```

首先是一些例行检查，常常称为“sanity check”，或者说“健康检查”、“卫生检查”。其中 109 行检查的是区间的大小既不为 0，也不能太大而越出了 32 位地址空间的限制。物理地址 0xa0000 至 0x100000 用于 VGA 卡和 BIOS，这是在系统初始化时就映射好了的，不能侵犯到这个区间中去。121 行中的 high_memory 是在系统初始化时，根据检测到的物理内存大小设置的物理内存地址的上限（所对应的虚拟地址）。如果所要求的 phys_addr 小于这个上限的话，就表示与系统的物理内存有冲突了，除非相应的物理页面原来就是保留着的空洞。在通过这些检查以后，还要保证该物理地址是按页面边

PDF-XCHANGE Click to buy NOW! www.docu-track.com

界对齐的（136～138 行）。

完成了这些准备以后，这才“言归正传”。首先是要找到一片虚存地址区间。前面讲过，这片区间属于内核，而不属于任何一个特定的进程，所以不是在某个进程的 mm_struct 结构中的虚存区间队列中去寻找，而是从属于内核的虚存区间队列中去寻找。函数 get_vm_area()是在 mm/vmalloc.c 中定义的：

==================== mm/vmalloc.c 168 201 ====================
[ioremap()>__ioremap()>get_vm_area()]

```
struct vm_struct * get_vm_area(unsigned long size, unsigned long flags)
{
	unsigned long addr;
	struct vm_struct **p, *tmp, *area;

	area = (struct vm_struct *) kmalloc(sizeof(*area), GFP_KERNEL);
	if (!area)
		return NULL;
	size += PAGE_SIZE;
	addr = VMALLOC_START;
	write_lock(&vmlist_lock);
	for (p = &vmlist; (tmp = *p) ; p = &tmp->next) {
		if ((size + addr) < addr) {
			write_unlock(&vmlist_lock);
			kfree(area);
			return NULL;
		}
		if (size + addr < (unsigned long) tmp->addr)
			break;
		addr = tmp->size + (unsigned long) tmp->addr;
		if (addr > VMALLOC_END-size) {
			write_unlock(&vmlist_lock);
			kfree(area);
			return NULL;
		}
	}
	area->flags = flags;
	area->addr = (void *)addr;
	area->size = size;
	area->next = *p;
	*p = area;
	write_unlock(&vmlist_lock);
	return area;
}
```

内核为自己保持一个虚存区间队列 vmlist，这是由一串 vm_struct 数据结构组成的一个单链队列。这里的 vm_struct 和 vmlist 都是由内核专用的。vm_struct 从概念上说类似于供进程使用的 vm_area_struct，但要简单得多，定义于 include/linux/vmalloc.h 和 mm/vmalloc.c 中：

==================== include/linux/vmalloc.h 14 19 ====================

```
struct vm_struct {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        unsigned long flags;
        void * addr;
        unsigned long size;
        struct vm_struct * next;
};
```

==================== mm/vmalloc.c 18 18 ====================

```
struct vm_struct * vmlist;
```

以前讲过，内核使用的系统空间虚拟地址与物理地址间存在着一种简单的映射关系，只要在物理地址上加上一个 3GB 的偏移量就得到了内核的虚拟地址。而变量 high_memory 标志着具体物理内存的上限所对应的虚拟地址，这是在系统初始化时设置好的。当内核需要一片虚存地址空间时，就从这个地址以下 8MB 处分配。为此，在 include/asm-i386/pgtable.h 中定义了 VMALLOC_START 等有关的常数：

==================== include/asm-i386/pgtable.h 132 143 ====================

```
132 /* Just any arbitrary offset to the start of the vmalloc VM area: the
133  * current 8MB value just means that there will be a 8MB "hole" after the
134  * physical memory until the kernel virtual memory starts.  That means that
135  * any out-of-bounds memory accesses will hopefully be caught.
136  * The vmalloc() routines leaves a hole of 4kB between each vmalloced
137  * area for the same reason. ;)
138  */
139 #define VMALLOC_OFFSET (8*1024*1024)
140 #define VMALLOC_START  (((unsigned long) high_memory + 2*VMALLOC_OFFSET-1) & \
141                             ~(VMALLOC_OFFSET-1))
142 #define VMALLOC_VMADDR(x) ((unsigned long)(x))
143 #define VMALLOC_END    (FIXADDR_START)
```

源代码中的注解对于为什么要留一个 8MB 的空洞，以及在每次分配虚存区间时也要留下一个页面的空洞（见 132 行）解释得很清楚：是为了便于捕捉可能的越界访问。

这里读者可能会有个问题，185 行的 if 语句检查的是当前的起始地址加上区间大小须小于下一个区间的起始地址，这是很好理解的。可是 176 行在区间大小上又加了一个页面作为空洞。这个空洞页面难道不可能与下一个区间的起始地址冲突吗？这里的奥妙在于 185 行判定的条件是“ <”而不是“ <=”，并且 size 和 addr 都是按页面边界对齐的，所以 185 行的条件已经隐含着其中有一个页面的宇洞。从 get_vm_area()成功返回时，就标志着所需要的一片虚存空间已经分配好了，从返回的数据结构可以得到这片空间的起始地址。下面就是建立映射的事了。

宏定义 VMALLOC_VMADDR 我们已经在前面看到过了，实际上不做什么事情，只是类型转换。函数 remap_area_pages()的代码也在 arch/i386/mm/ioremap.c 中：

==================== arch/i386/mm/ioremap.c 62 86 ====================

[ioremap()>__ioremap()>remap_area_pages()]

```
static int remap_area_pages(unsigned long address, unsigned long phys_addr,
                    unsigned long size, unsigned long flags)
{
        pgd_t * dir;
        unsigned long end = address + size;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
67
68        phys_addr -= address;
69        dir = pgd_offset(&init_mm, address);
70        flush_cache_all();
71        if (address >= end)
72             BUG();
73        do {
74             pmd_t *pmd;
75             pmd = pmd_alloc_kernel(dir, address);
76             if (!pmd)
77                  return -ENOMEM;
78             if (remap_area_pmd(pmd, address, end - address,
79                             phys_addr + address, flags))
80                  return -ENOMEM;
81             address = (address + PGDIR_SIZE) & PGDIR_MASK;
82             dir++;
83        } while (address && (address < end));
84        flush_tlb_all();
85        return 0;
86   }
```

我们讲过，每个进程的 task_struct 结构中都有一个指针指向 mm_struct 结构，从中可以找到相应的页面目录。但是，内核空间不属于任何一个特定的进程，所以单独设置了一个内核专用的 mm_struct，称为 init_mm。当然，内核也没有代表它的 task_struct 结构，所以 69 行根据起始地址从 init_mm 中找到所属的目录项，然后就根据区间的大小走遍所有涉及的目录项。这里的 68 行看似奇怪。从物理地址中减去虚拟地址得出一个负的位移量，这个位移量在 78～79 行又与虚拟地址相加，仍旧得到物理地址。由于在循环中虚拟地址 address 在变（见 81 行），物理地址也就相应而变。第 75 行的 pmd_alloc_kernel() 对于 i386 CPU 就是 pmd_alloc()，定义于 include/asm-i386/pgalloc.h：

==================== include/asm-i386/pgalloc.h 151 151 ====================

```
#define pmd_alloc_kernel    pmd_alloc
```

而 inline 函数 pmd_alloc()的定义则有两个，分别用于二级和三级映射。对于二级映射这个定义为（见 include/asm-i386/pgalloc-2level.h）：

==================== include/asm-i386/pgalloc-2level.h 16 21 ====================

[ioremap()>__ioremap()>remap_area_pages()>pmd_alloc()]

```
extern inline pmd_t * pmd_alloc(pgd_t *pgd, unsigned long address)
{
     if (!pgd)
          BUG();
     return (pmd_t *) pgd;
}
```

可见，对于 i386 的二级页式映射，只是把页面目录项当成中间目录项而已，与“分配”实际上毫无关系。即使对于采用了物理地址扩充（PAE）的 Pentium CPU，虽然实现三级映射，其作用也只是“找到”中间目录项而已，只有在中间目录项为空时才真的分配一个。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这样，remap_area_pages()中从 73 行开始的 do-while 循环，对涉及到的每个页面目录表项调用 remap_area_pmd()。而 remap_area_pmd()几乎完全一样，对涉及到的每个页面表（对 i386 的二级映射，每个中间目录项实际上就是一个页面表项，也对以理解为中间目录表的大小为 1）调用 remap_area_pte()，这也是在 arch/i386/mm/ioremap.c 中定义的：

==================== arch/i386/mm/ioremap.c 15 37 ====================
[ioremap()>__ioremap()>remap_area_pages()>remap_area_pmd()>remap_area_pte()]

```
15   static inline void remap_area_pte(pte_t * pte, unsigned long address, unsigned long size,
16        unsigned long phys_addr, unsigned long flags)
17   {
18        unsigned long end;
19
20        address &= ~PMD_MASK;
21        end = address + size;
22        if (end > PMD_SIZE)
23             end = PMD_SIZE;
24        if (address >= end)
25             BUG();
26        do {
27             if (!pte_none(*pte)) {
28                  printk("remap_area_pte: page already exists\n");
29                  BUG();
30             }
31             set_pte(pte, mk_pte_phys(phys_addr, __pgprot(_PAGE_PRESENT | _PAGE_RW |
32                            _PAGE_DIRTY | _PAGE_ACCESSED | flags)));
33             address += PAGE_SIZE;
34             phys_addr += PAGE_SIZE;
35             pte++;
36        } while (address && (address < end));
37   }
```

这里只是简单地在循环中设置页面表中所有涉及的页面表项（31 行）。每个表项都被预设成_PAGE_DIRTY、_PAGE_ACCESSED 和_PAGE_PRESENTED。

在 kswapd 换出页面的情景中，我们已经看到 kswapd 定期地、循环地、依次地从 task 结构队列中找出占用内存页面最多的进程，然后就对该进程调用 swap_out_mm()换出一些页面。而内核的 mm_struct 结构 init_mm 是单独的，从任何一个进程的 task 结构中都到达不了 init_mm。所以，kswapd 根本就看不到 init_mm 中的虚存区间，这些区间的页面就自然不会被换出而长驻于内存。

2.12 系统调用 brk()

尽管“可见度”不高，brk()也许是最常使用的系统调用了，用户进程通过它向内核申请空间。人们常常并不意识到在调用 brk()，原因在于很少有人会直接使用系统调用 brk()向系统申请空间，而总是通过像 malloc()一类的 C 语言库函数（或语言成分，如 C++中的 new）间接地用到 brk()。如果把 malloc()想像成零售，brk()则是批发。库函数 malloc()为用户进程（malloc 本身就是该进程的一部分）维持一个

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

小仓库，当进程需要使用更多的内存空间时就向小仓库要，小仓库中存量不足时就通过 brk()向内核批发。

前面讲过，每个进程拥有 3G 字节的用户虚存空间。但是，这并不意味着用户进程在这 3G 字节的范围里可以任意使用，因为虚存空间最终得映射到某个物理存储空间（内存或磁盘空间），才真正可以使用，而这种映射的建立和管理则由内核处理。所谓向内核申请一块空间，是指请求内核分配一块虚存区间和相应的若干物理页面，并建立起映射关系。由于每个进程的虚存空间都很大（3G），而实际需要使用的又很小，内核不可能在创建进程时就为整个虚存空间都分配好相应的物理空间并建立映射，而只能是需要用多少才“分配”多少。

那么，内核怎样管理每个进程的 3G 字节虚存空间呢？粗略地说，用户程序经过编译、连接形成的映象文件中有一个代码段和一个数据段（包括 data 段和 bss 段），其中代码段在下，数据段在上。数据段中包括了所有静态分配的数据空间，包括全局变量和说明为 static 的局部变量。这些空间是进程所必须的基本要求，所以内核在建立一个进程的运行映象时就分配好这些空间，包括虚存地址区间和物理页面，并建立好二者间的映射。除此之外，堆栈使用的空间也属于基本要求，所以也是在建立进程时就分配好的（但可以扩充）。所不同的是，堆栈中间安置在虚存空间的顶部，运行时由顶向下延伸；代码段和数据段则在底部（注意，不要与 X86 系统结构中由段寄存器建立的“代码段”及“数据段”相混淆），在运行时并不向上伸展。而从数据段的顶部 end_data 到堆栈段地址的下沿这个中间区域则是一个巨大的空洞，这就是可以在运行时动态分配的空间。最初，这个动态分配空间是从进程的 end_data 开始的，这个地址为内核和进程所共知。以后，每次动态分配一块“内存”，这个边界就往上推进一段距离，同时内核和进程都要记下当前的边界在哪里。在进程这一边由 malloc()或类似的库函数管理，而在内核中则将当前的边界记录在进程的 mm_struct 结构中。具体地说，mm_struct 结构中有一个成分 brk，表示动态分配区当前的底部。当一个进程需要分配内存时，将要求的大小与其当前的动态分配区底部边界相加，所得的就是所要求的新边界，也就是 brk()调用时的参数 brk。当内核能满足要求时，系统调用 brk()返回 0，此后新旧两个边界之间的虚存地址就都可以使用了。当内核发现无法满足要求（例如物理空间已经分配完），或者发现新的边界已经过了逼近设于顶部的堆栈时，就拒绝分配而返回-1。

系统调用 brk()在内核中的实现为 sys_brk()，其代码在 mm/mmap.c 中。这个函数既可以用来分配空间，即把动态分配区底部的边界往上推；也可以用来释放，即归还空间。因此，它的代码也大致上可以分成两部分。我们先读第一部分：

==================== mm/mmap.c 113 141 ====================

[sys_brk()]

```
/*
 *  sys_brk() for the most part doesn't need the global kernel
 *  lock, except when an application is doing something nasty
 *  like trying to un-brk an area that has already been mapped
 *  to a regular file.  in this case, the unmapping will need
 *  to invoke file system routines that need the global lock.
 */
asmlinkage unsigned long sys_brk(unsigned long brk)
{
	unsigned long rlim, retval;
	unsigned long newbrk, oldbrk;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct mm_struct *mm = current->mm;

    down(&mm->mmap_sem);

    if (brk < mm->end_code)
        goto out;
    newbrk = PAGE_ALIGN(brk);
    oldbrk = PAGE_ALIGN(mm->brk);
    if (oldbrk == newbrk)
        goto set_brk;

    /* Always allow shrinking brk. */
    if (brk <= mm->brk) {
        if (!do_munmap(mm, newbrk, oldbrk-newbrk))
            goto set_brk;
        goto out;
    }

```

参数 brk 表示所要求的新边界，这个边界不能低于代码段的终点，并且必须与页面大小对齐。如果新边界低于老边界，那就不是申请分配空间，而是释放空间，所以通过 do_munmap()解除一部分区间的映射，这是个重要的函数。其代码在 mm/mmap.c 中：

==================== mm/mmap.c 664 696 ====================

[sys_brk()>do_munmap()]

```
/* Munmap is split into 2 main parts -- this part which finds
 * what needs doing, and the areas themselves, which do the
 * work.  This now handles partial unmappings.
 * Jeremy Fitzhardine <jeremy@sw.oz.au>
 */
int do_munmap(struct mm_struct *mm, unsigned long addr, size_t len)
{
    struct vm_area_struct *mpnt, *prev, **npp, *free, *extra;

    if ((addr & ~PAGE_MASK) || addr > TASK_SIZE || len > TASK_SIZE-addr)
        return -EINVAL;

    if ((len = PAGE_ALIGN(len)) == 0)
        return -EINVAL;

    /* Check if this memory area is ok - put it on the temporary
     * list if so..  The checks here are pretty simple --
     * every area affected in some way (by any overlap) is put
     * on the list.  If nothing is put on, nothing is affected.
     */
    mpnt = find_vma_prev(mm, addr, &prev);
    if (!mpnt)
        return 0;
    /* we have  addr < mpnt->vm_end  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    if (mpnt->vm_start >= addr+len)
        return 0;

    /* If we'll make "hole", check the vm areas limit */
    if ((mpnt->vm_start < addr && mpnt->vm_end > addr+len)
        && mm->map_count >= MAX_MAP_COUNT)
        return -ENOMEM;

```

函数 find_vma_prev()的作用与以前在“几个重要的数据结构和函数”一节中读过的 find_vma()基本相同，它扫描当前进程用户空间的 vm_area_struct 结构链表或 AVL 树，试图找到结束地址高于 addr 的第一个区间，如果找到，则函数返回该区间的 vm_area_struct 结构指针。不同的是，它同时还通过参数 prev 返回其前一区间结构的指针。等一下我们就将看到为什么需要这个指针。如果返回的指针为 0，或者该区间的起始地址也高于 addr+len，那就表示想要解除映射的那部分空间原来就没有映射，所以直接返回 0。如果这部分空间落在某个区间的中间，则在解除这部分空间的映射以后会造成一个空洞而使原来的区间一分为二。可是，一个进程可以拥有的虚存区间的数量是有限制的，所以若这个数量达到了上限 MAX_MAP_COUNT，就不再允许这样的操作。

我们继续往下看：

===================== mm/mmap.c 697 717 =====================

[sys_brk()>do_munmap()]

```
    /*
     * We may need one additional vma to fix up the mappings ...
     * and this is the last chance for an easy error exit.
     */
    extra = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!extra)
        return -ENOMEM;

    npp = (prev ? &prev->vm_next : &mm->mmap);
    free = NULL;
    spin_lock(&mm->page_table_lock);
    for ( ; mpnt && mpnt->vm_start < addr+len; mpnt = *npp) {
        *npp = mpnt->vm_next;
        mpnt->vm_next = free;
        free = mpnt;
        if (mm->mmap_avl)
            avl_remove(mpnt, &mm->mmap_avl);
    }
    mm->mmap_cache = NULL; /* Kill the cache. */
    spin_unlock(&mm->page_table_lock);

```

由于解除一部分空间的映射有可能使原来的区间一分为二，所以这里先分配好一个空白的 vm_area_struct 结构 extra。另一方面，要解除映射的那部分空间也有可能跨越好几个区间，所以通过一

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

个 for 循环把所有涉及的区间都转移到一个临时队列 free 中，如果建立了 AVL 树，则也要把这些区间的 vm_area_struct 结构从 AVL 树中删除。以前讲过，mm_struct 结构中的指针 mmap_cache 指向上一次 find_vma()操作的对象，因为对虚存区间的操作往往是有连续性的（见 find_vma()的代码），而现在用户空间的结构有了变化，多半已经打破了这种连续性，所以把它清成 0。至此，已经完成了所有的准备，下面就要具体解除映射了。

==================== mm/mmap.c 718 762 ====================

[sys_brk()>do_munmap()]

```
	/* Ok - we have the memory areas we should free on the 'free' list,
	 * so release them, and unmap the page range..
	 * If the one of the segments is only being partially unmapped,
	 * it will put new vm_area_struct(s) into the address space.
	 * In that case we have to be careful with VM_DENYWRITE.
	 */
	while ((mpnt = free) != NULL) {
		unsigned long st, end, size;
		struct file *file = NULL;

		free = free->vm_next;

		st = addr < mpnt->vm_start ? mpnt->vm_start : addr;
		end = addr+len;
		end = end > mpnt->vm_end ? mpnt->vm_end : end;
		size = end - st;

		if (mpnt->vm_flags & VM_DENYWRITE &&
		    (st != mpnt->vm_start || end != mpnt->vm_end) &&
		    (file = mpnt->vm_file) != NULL) {
			atomic_dec(&file->f_dentry->d_inode->i_writecount);
		}
		remove_shared_vm_struct(mpnt);
		mm->map_count--;

		flush_cache_range(mm, st, end);
		zap_page_range(mm, st, size);
		flush_tlb_range(mm, st, end);

		/*
		 * Fix the mapping, and free the old area if it wasn't reused.
		 */
		extra = unmap_fixup(mm, mpnt, st, size, extra);
		if (file)
			atomic_inc(&file->f_dentry->d_inode->i_writecount);
	}

	/* Release the extra vma struct if it wasn't used */
	if (extra)
		kmem_cache_free(vm_area_cachep, extra);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
758
759         free_pgtables(mm, prev, addr, addr+len);
760
761         return 0;
762     }
```

这里通过一个 while 循环逐个处理所涉及的区间，这些区间的 vm_area_struct 结构都链接在一个临时的队列 free 中。在下一节中读者将看到，一个进程可以通过系统调用 mmap()将一个文件的内容映射到其用户空间的某个区间，然后就像访问内存一样来访问这个文件。但是，如果这个文件同时又被别的进程打开，并通过常规的文件操作访问，则在二者对此文件的两种不同形式的写操作之间要加以互斥。如果要解除映射的只是这样的区间的一部分（735～737 行），那就相当于对此区间的写操作，所以要递减该文件的 inode 结构中的一个计数器 i_writecount，以保证互斥，到操作完成以后再予恢复（751～752 行）。同时，还要通过 remove_shared_vm_struct()看看所处理的区间是否是这样的区间，如果是，就将其 vm_area_struct 结构从目标文件的 inode 结构内的 i_mapping 队列中脱链。

代码中的 zap_page_range()解除若干连续页面的映射，并且释放所映射的内存页面，或对交换设备上物理页面的引用，这才是我们在这里所主要关心的。其代码在 mm/memory.c 中：

==================== mm/memory.c 348 383 ====================

[sys_brk()>do_munmap()>zap_page_range()]

```
/*
 * remove user pages in a given range.
 */
void zap_page_range(struct mm_struct *mm, unsigned long address, unsigned long size)
{
	pgd_t * dir;
	unsigned long end = address + size;
	int freed = 0;

	dir = pgd_offset(mm, address);

	/*
	 * This is a long-lived spinlock. That's fine.
	 * There's no contention, because the page table
	 * lock only protects against kswapd anyway, and
	 * even if kswapd happened to be looking at this
	 * process we _want_ it to get stuck.
	 */
	if (address >= end)
		BUG();
	spin_lock(&mm->page_table_lock);
	do {
		freed += zap_pmd_range(mm, dir, address, end - address);
		address = (address + PGDIR_SIZE) & PGDIR_MASK;
		dir++;
	} while (address && (address < end));
	spin_unlock(&mm->page_table_lock);
	/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
376       * Update rss for the mm_struct (not necessarily current->mm)
377       * Notice that rss is an unsigned long.
378       */
379      if (mm->rss > freed)
380           mm->rss -= freed;
381      else
382           mm->rss = 0;
383  }
```

这个函数解除一块虚存区间的页面映射。首先通过 pgd_offset()在第一层页面目录中找到起始地址所属的目录项，然后就通过一个 do-while 循环从这个目录项开始处理涉及的所有目录项。

==================== include/asm-i386/pgtable.h 312 312 ====================

```
312  #define pgd_index(address) ((address >> PGDIR_SHIFT) & (PTRS_PER_PGD-1))
```

==================== include/asm-i386/pgtable.h 316 316 ====================

```
316  #define pgd_offset(mm, address) ((mm)->pgd+pgd_index(address))
```

对于涉及的每一个目录项，通过 zap_pmd_range()处理第二层的中间目录表。

==================== mm/memory.c 321 346 ====================

[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()]

```
321  static inline int zap_pmd_range(struct mm_struct *mm, pgd_t * dir, unsigned long address, unsigned long
size)
322  {
323      pmd_t * pmd;
324      unsigned long end;
325      int freed;
326
327      if (pgd_none(*dir))
328           return 0;
329      if (pgd_bad(*dir)) {
330           pgd_ERROR(*dir);
331           pgd_clear(dir);
332           return 0;
333      }
334      pmd = pmd_offset(dir, address);
335      address &= ~PGDIR_MASK;
336      end = address + size;
337      if (end > PGDIR_SIZE)
338           end = PGDIR_SIZE;
339      freed = 0;
340      do {
341           freed += zap_pte_range(mm, pmd, address, end - address);
342           address = (address + PMD_SIZE) & PMD_MASK;
343           pmd++;
344      } while (address < end);
345      return freed;
346  }
```

同样，先通过 pmd_offset()，在第二层目录表中找到起始目录项。对于采用二级映射的 i386 结构，

中间目录表这一层是空的。pmd_offset()的定义在 include/asm-i386/pgtable-2level.h 中：

==================== include/asm-i386/pgtable-2level.h 53 56 ====================

```
extern inline pmd_t * pmd_offset(pgd_t * dir, unsigned long address)
{
	return (pmd_t *) dir;
}
```

可见，pmd_offset()把指向第一层目录项的指针原封不动地作为指向中间目录项的指针返回来了，也就是说把第一层目录当成了中间目录。所以，对于二级映射，zap_pmd_range()在某种意义上只是把 zap_page_range()所做的事重复了一遍。不过，这一次重复调用的是 zap_pte_range()，处理的是底层的页面映射表了。

==================== mm/memory.c 289 319 ====================

[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()>zap_pte_range()]

```
static inline int zap_pte_range(struct mm_struct *mm, pmd_t * pmd, unsigned long address, unsigned long size)
{
	pte_t * pte;
	int freed;

	if (pmd_none(*pmd))
		return 0;
	if (pmd_bad(*pmd)) {
		pmd_ERROR(*pmd);
		pmd_clear(pmd);
		return 0;
	}
	pte = pte_offset(pmd, address);
	address &= ~PMD_MASK;
	if (address + size > PMD_SIZE)
		size = PMD_SIZE - address;
	size >>= PAGE_SHIFT;
	freed = 0;
	for (;;) {
		pte_t page;
		if (!size)
			break;
		page = ptep_get_and_clear(pte);
		pte++;
		size--;
		if (pte_none(page))
			continue;
		freed += free_pte(page);
	}
	return freed;
}
```

还是先找到在给定页面表中的起始表项，与 pte_offset()有关的定义在 include/asm-i386/pgtable.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/asm-i386/pgtable.h 324 328 ====================

```
/* Find an entry in the third-level page table.. */
#define __pte_offset(address) \
        ((address >> PAGE_SHIFT) & (PTRS_PER_PTE - 1))
#define pte_offset(dir, address) ((pte_t *) pmd_page(*(dir)) + \
            __pte_offset(address))
```

然后就是在一个 for 循环中，对需要解除映射的页面调用 ptep_get_and_clear()将页面表项清成 0：

==================== include/asm-i386/pgtable-2level.h 57 57 ====================

```
#define ptep_get_and_clear(xp)   __pte(xchg(&(xp)->pte_low, 0))
```

最后通过 free_pte()解除对内存页面以及盘上页面的使用，这个函数的代码在 mm/memory.c 中：

==================== mm/memory.c 259 279 ====================

[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()>zap_pte_range()>free_pte()]

```
/*
 * Return indicates whether a page was freed so caller can adjust rss
 */
static inline int free_pte(pte_t pte)
{
    if (pte_present(pte)) {
        struct page *page = pte_page(pte);
        if ((!VALID_PAGE(page)) || PageReserved(page))
            return 0;
        /*
         * free_page() used to be able to clear swap cache
         * entries.  We may now have to do it manually.
         */
        if (pte_dirty(pte) && page->mapping)
            set_page_dirty(page);
        free_page_and_swap_cache(page);
        return 1;
    }
    swap_free(pte_to_swp_entry(pte));
    return 0;
}
```

如果页面表项表明在解除映射前页面就已不在内存，则当前进程对该内存页面的使用已经解除，所以只需调用 swap_free()解除对交换设备上的“盘上页面”的使用。当然，swap_free()首先是递减盘上页面的使用计数，只有当这个计数达到 0 时才真正地释放了这个盘上页面。如果当前进程是这个盘上页面的最后一个用户（或惟一的用户），则该计数递减后为 0。反之，则要通过 free_page_and_swap_cache()解除对盘上页面和内存页面二者的使用。此外，如果页面在最近一次 try_to_swap_out()以后已被写过，则还要通过 set_page_dirty()设置该页面 page 结构中的 PG_dirty 标志位，并在相应的 address_space 结构中将其移入 dirty_pages 队列。函数 free_page_and_swap_cache()的代码在 mm/swap_state.c 中：

==================== mm/swap_state.c 133 150 ====================

[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()>zap_pte_range()>free_pte()>free_page_and_swap_cache()]

```
/*
 * Perform a free_page(), also freeing any swap cache associated with
 * this page if it is the last user of the page. Can not do a lock_page,
 * as we are holding the page_table_lock spinlock.
 */
void free_page_and_swap_cache(struct page *page)
{
	/* 
	 * If we are the only user, then try to free up the swap cache. 
	 */
	if (PageSwapCache(page) && !TryLockPage(page)) {
		if (!is_page_shared(page)) {
			delete_from_swap_cache_nolock(page);
		}
		UnlockPage(page);
	}
	page_cache_release(page);
}
```

以前讲过，一个有用户空间映射、可换出的内存页面（确切地说是它的 page 数据结构），同时在三个队列中。一是通过其队列头 list 链入某个换入／换出队列，即相应 address_space 结构中的 clean_pages、dirty_pages 以及 locked_pages 三个队列之一；二是通过其队列头 lru 链入某个 LRU 队列，即 active_list、inactive_dirty_list 或者某个 inactive_clean_list 之一；最后就是通过指针 next_hash 链入一个杂凑队列。当一个页面在某个换入／换出队列中时，其 page 结构中的 PG_swap_cache 标志位为 1，如果当前进程是这个页面的最后一个用户（或惟一用户），此时便要调用 delete_from_swap_cache_nolock()将页面从上述队列中脱离出来。

==================== mm/swap_state.c 103 120 ====================
[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()>zap_pte_range()>free_pte()>free_page_and_swap_cache()>delete_from_swap_cache_nolock()]

```
/*
 * This will never put the page into the free list, the caller has
 * a reference on the page.
 */
void delete_from_swap_cache_nolock(struct page *page)
{
	if (!PageLocked(page))
		BUG();

	if (block_flushpage(page, 0))
		lru_cache_del(page);

	spin_lock(&pagecache_lock);
	ClearPageDirty(page);
	__delete_from_swap_cache(page);
	spin_unlock(&pagecache_lock);
	page_cache_release(page);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
120  }
```

先通过 block_flushpage()把页面的内容“冲刷”到块设备上，不过实际上这种冲刷仅在页面来自一个映射到用户空间的文件时才进行，因为对于交换设备上的页面，此时的内容已经没有意义了。完成了冲刷以后，就通过 lru_cache_del()将页面从其所在的 LRU 队列中脱离出来。然后，再通过__delete_from_swap_cache()，使页面脱离其它两个队列。

==================== mm/swap_state.c 86 101 ====================
[sys_brk()>do_munmap()>zap_page_range()>zap_pmd_range()>zap_pte_range()>free_pte()>free_page_and_swap_cache()>delete_from_swap_cache_nolock()>__delete_from_swap_cache()]

```
/*
 * This must be called only on pages that have
 * been verified to be in the swap cache.
 */
void __delete_from_swap_cache(struct page *page)
{
	swp_entry_t entry;

	entry.val = page->index;

#ifdef SWAP_CACHE_INFO
	swap_cache_del_total++;
#endif
	remove_from_swap_cache(page);
	swap_free(entry);
}
```

这里的 remove_from_swap_cache()将页面的 page 结构从换入／换出队列和杂凑队列中脱离出来。然后，也是通过 swap_free()释放盘上页面，回到 delete_from_swap_cache_nolock()。最后是 page_cache_release()，即递减 page 结构中的使用计数。由于当前进程是页面的最后一个用户，并且在解除映射之前页面在内存中（见上面 free_pte()中的 264 行），所以页面的使用计数应该是 2，这里（119 行）调用了一次 page_cache_release()就使其变成了 1。再返回到 free_page_and_swap_cache()中，这里（149 行）又调用了一次 page_cache_release()，这一次就使其变成了 0，于是就最终把页面释放，让它回到了空闲页面队列中。

当回到 do_munmap()中的时候，已经完成了对一个虚存区间的操作。此时，一方面要对虚存区间的 vm_area_struct 数据结构和进程的 mm_struct 数据结构作出调整，以反映已经发生的变化，如果整个区间都解除了映射，则要释放原有的 vm_area_struct 数据结构；另一方面原来的区间还可能要一分为二，因而需要插入一个新的 vm_area_struct 数据结构。这些操作是由 unmap_fixup()完成的，其代码在 mm/mmap.c 中：

==================== mm/mmap.c 516 604 ====================
[sys_brk()>do_munmap()>unmap_fixup()]

```
/* Normal function to fix up a mapping
 * This function is the default for when an area has no specific
 * function.  This may be used as part of a more specific routine.
 * This function works out what part of an area is affected and
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * adjusts the mapping information.  Since the actual page
 * manipulation is done in do_mmap(), none need be done here,
 * though it would probably be more appropriate.
 *
 * By the time this function is called, the area struct has been
 * removed from the process mapping list, so it needs to be
 * reinserted if necessary.
 *
 * The 4 main cases are:
 *    Unmapping the whole area
 *    Unmapping from the start of the segment to a point in it
 *    Unmapping from an intermediate point to the end
 *    Unmapping between to intermediate points, making a hole.
 *
 * Case 4 involves the creation of 2 new areas, for each side of
 * the hole.  If possible, we reuse the existing area rather than
 * allocate a new one, and the return indicates whether the old
 * area was reused.
 */
static struct vm_area_struct * unmap_fixup(struct mm_struct *mm,
        struct vm_area_struct *area, unsigned long addr, size_t len,
        struct vm_area_struct *extra)
{
        struct vm_area_struct *mpnt;
        unsigned long end = addr + len;

        area->vm_mm->total_vm -= len >> PAGE_SHIFT;
        if (area->vm_flags & VM_LOCKED)
                area->vm_mm->locked_vm -= len >> PAGE_SHIFT;

        /* Unmapping the whole area. */
        if (addr == area->vm_start && end == area->vm_end) {
                if (area->vm_ops && area->vm_ops->close)
                        area->vm_ops->close(area);
                if (area->vm_file)
                        fput(area->vm_file);
                kmem_cache_free(vm_area_cachep, area);
                return extra;
        }

        /* Work out to one of the ends. */
        if (end == area->vm_end) {
                area->vm_end = addr;
                lock_vma_mappings(area);
                spin_lock(&mm->page_table_lock);
        } else if (addr == area->vm_start) {
                area->vm_pgoff += (end - area->vm_start) >> PAGE_SHIFT;
                area->vm_start = end;
                lock_vma_mappings(area);
```

```
        spin_lock(&mm->page_table_lock);
    } else {
    /* Unmapping a hole: area->vm_start < addr <= end < area->vm_end */
        /* Add end mapping -- leave beginning for below */
        mpnt = extra;
        extra = NULL;

        mpnt->vm_mm = area->vm_mm;
        mpnt->vm_start = end;
        mpnt->vm_end = area->vm_end;
        mpnt->vm_page_prot = area->vm_page_prot;
        mpnt->vm_flags = area->vm_flags;
        mpnt->vm_raend = 0;
        mpnt->vm_ops = area->vm_ops;
        mpnt->vm_pgoff = area->vm_pgoff + ((end - area->vm_start) >> PAGE_SHIFT);
        mpnt->vm_file = area->vm_file;
        mpnt->vm_private_data = area->vm_private_data;
        if (mpnt->vm_file)
            get_file(mpnt->vm_file);
        if (mpnt->vm_ops && mpnt->vm_ops->open)
            mpnt->vm_ops->open(mpnt);
        area->vm_end = addr;   /* Truncate area */

        /* Because mpnt->vm_file == area->vm_file this locks
         * things correctly.
         */
        lock_vma_mappings(area);
        spin_lock(&mm->page_table_lock);
        __insert_vm_struct(mm, mpnt);
    }

    __insert_vm_struct(mm, area);
    spin_unlock(&mm->page_table_lock);
    unlock_vma_mappings(area);
    return extra;
}
```

我们把这段代码留给读者。最后，当循环结束之时，由于已经解除了一些页面的映射，有些页面映射表可能整个都已经空白，对于这样的页面表（所占的页面）也要加以释放。这是由 free_pgtables() 完成的。我们也把它的代码留给读者（mm/mmap.c）。

==================== mm/mmap.c 606 662 ====================

[sys_brk()>do_munmap()>free_pgtables()]

```
/*
 * Try to free as many page directory entries as we can,
 * without having to work very hard at actually scanning
 * the page tables themselves.
 *
 * Right now we try to free page tables if we have a nice
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * PGDIR-aligned area that got free'd up. We could be more
 * granular if we want to, but this is fast and simple,
 * and covers the bad cases.
 *
 * "prev", if it exists, points to a vma before the one
 * we just free'd - but there's no telling how much before.
 */
static void free_pgtables(struct mm_struct * mm, struct vm_area_struct *prev,
    unsigned long start, unsigned long end)
{
    unsigned long first = start & PGDIR_MASK;
    unsigned long last = end + PGDIR_SIZE - 1;
    unsigned long start_index, end_index;

    if (!prev) {
        prev = mm->mmap;
        if (!prev)
            goto no_mmaps;
        if (prev->vm_end > start) {
            if (last > prev->vm_start)
                last = prev->vm_start;
            goto no_mmaps;
        }
    }
    for (;;) {
        struct vm_area_struct *next = prev->vm_next;

        if (next) {
            if (next->vm_start < start) {
                prev = next;
                continue;
            }
            if (last > next->vm_start)
                last = next->vm_start;
        }
        if (prev->vm_end > first)
            first = prev->vm_end + PGDIR_SIZE - 1;
        break;
    }
no_mmaps:
    /*
     * If the PGD bits are not consecutive in the virtual address, the
     * old method of shifting the VA >> by PGDIR_SHIFT doesn't work.
     */
    start_index = pgd_index(first);
    end_index = pgd_index(last);
    if (end_index > start_index) {
        clear_page_tables(mm, start_index, end_index - start_index);
        flush_tlb_pgtables(mm, first & PGDIR_MASK, last & PGDIR_MASK);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    }
}
```

回到 sys_brk()的代码中，我们已经完成了通过 sys_brk()释放空间的情景分析。

如果新边界高于老边界，就表示要分配空间，这就是 sys_brk()的后一部分。我们继续往下看（mm/mmap.c）：

==================== mm/mmap.c 142 164 ====================

[sys_brk()]

```
    /* Check against rlimit.. */
    rlim = current->rlim[RLIMIT_DATA].rlim_cur;
    if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
        goto out;

    /* Check against existing mmap mappings. */
    if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
        goto out;

    /* Check if we have enough memory.. */
    if (!vm_enough_memory((newbrk-oldbrk) >> PAGE_SHIFT))
        goto out;

    /* Ok, looks good - let it rip. */
    if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
        goto out;
set_brk:
    mm->brk = brk;
out:
    retval = mm->brk;
    up(&mm->mmap_sem);
    return retval;
}
```

首先检查对进程的资源限制，如果所要求的新边界使数据段的大小超过了对当前进程的限制，就拒绝执行。此外，还要通过 find_vma_intersection()，检查所要求的那部分空间是否与已经存在的某一区间相冲突，这个 inline 函数的代码在 include/linux/mm.h 中：

==================== include/linux/mm.h 511 520 ====================

[sys_brk()>find_vma_intersection()]

```
/* Look up the first VMA which intersects the interval start_addr..end_addr-1,
   NULL if none.  Assume start_addr < end_addr. */
static inline struct vm_area_struct * find_vma_intersection(struct mm_struct * mm, unsigned long start_addr, unsigned long end_addr)
{
    struct vm_area_struct * vma = find_vma(mm,start_addr);

    if (vma && end_addr <= vma->vm_start)
        vma = NULL;
    return vma;
```

```
}
```

这里的 start_addr 是老的边界，如果 find_vma()返回一个非 0 指针，就表示在它之上已经有了一个已映射区间，因此有冲突的可能。此时新的边界 end_addr 必须落在这个区间的起点之下，也就是让从 start_addr 到 end_addr 这块空间落在空洞中，否则便是有了冲突。在查明了不存在冲突以后，还要通过 vm_enough_memoty()看看系统中是否有足够的空闲内存页面。

==================== mm/mmap.c 41 67 ====================

[sys_brk()>vm_enough_memoty()]

```
/* Check that a process has enough memory to allocate a
 * new virtual mapping.
 */
int vm_enough_memory(long pages)
{
	/* Stupid algorithm to decide if we have enough memory: while
	 * simple, it hopefully works in most obvious cases.. Easy to
	 * fool it, but this should catch most mistakes.
	 */
	/* 23/11/98 NJC: Somewhat less stupid version of algorithm,
	 * which tries to do "TheRightThing".  Instead of using half of
	 * (buffers+cache), use the minimum values.  Allow an extra 2%
	 * of num_physpages for safety margin.
	 */

	long free;

	  /* Sometimes we want to use more memory than we have. */
	if (sysctl_overcommit_memory)
	    return 1;

	free = atomic_read(&buffermem_pages);
	free += atomic_read(&page_cache_size);
	free += nr_free_pages();
	free += nr_swap_pages;
	return free > pages;
}
```

通过了这些检查，接着就是操作的主体 do_brk()了。这个函数的代码在 mm/mmap.c 中：

==================== mm/mmap.c 775 861 ====================

[sys_brk()>do_brk()]

```
/*
 *  this is really a simplified "do_mmap".  it only handles
 *  anonymous maps.  eventually we may be able to do some
 *  brk-specific accounting here.
 */
unsigned long do_brk(unsigned long addr, unsigned long len)
{
	struct mm_struct * mm = current->mm;
```

```
	struct vm_area_struct * vma;
	unsigned long flags, retval;

	len = PAGE_ALIGN(len);
	if (!len)
		return addr;

	/*
	 * mlock MCL_FUTURE?
	 */
	if (mm->def_flags & VM_LOCKED) {
		unsigned long locked = mm->locked_vm << PAGE_SHIFT;
		locked += len;
		if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
			return -EAGAIN;
	}

	/*
	 * Clear old maps.  this also does some error checking for us
	 */
	retval = do_munmap(mm, addr, len);
	if (retval != 0)
		return retval;

	/* Check against address space limits *after* clearing old maps... */
	if ((mm->total_vm << PAGE_SHIFT) + len
	    > current->rlim[RLIMIT_AS].rlim_cur)
		return -ENOMEM;

	if (mm->map_count > MAX_MAP_COUNT)
		return -ENOMEM;

	if (!vm_enough_memory(len >> PAGE_SHIFT))
		return -ENOMEM;

	flags = vm_flags(PROT_READ|PROT_WRITE|PROT_EXEC,
				MAP_FIXED|MAP_PRIVATE) | mm->def_flags;

	flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;


	/* Can we just expand an old anonymous mapping? */
	if (addr) {
		struct vm_area_struct * vma = find_vma(mm, addr-1);
		if (vma && vma->vm_end == addr && !vma->vm_file &&
		    vma->vm_flags == flags) {
			vma->vm_end = addr + len;
			goto out;
		}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
832      }
833
834
835      /*
836       * create a vma struct for an anonymous mapping
837       */
838      vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
839      if (!vma)
840           return -ENOMEM;
841
842      vma->vm_mm = mm;
843      vma->vm_start = addr;
844      vma->vm_end = addr + len;
845      vma->vm_flags = flags;
846      vma->vm_page_prot = protection_map[flags & 0x0f];
847      vma->vm_ops = NULL;
848      vma->vm_pgoff = 0;
849      vma->vm_file = NULL;
850      vma->vm_private_data = NULL;
851
852      insert_vm_struct(mm, vma);
853
854  out:
855      mm->total_vm += len >> PAGE_SHIFT;
856      if (flags & VM_LOCKED) {
857           mm->locked_vm += len >> PAGE_SHIFT;
858           make_pages_present(addr, addr + len);
859      }
860      return addr;
861  }
```

参数 addr 为需要建立映射的新区间的起点，1en 则为区间的长度。前面我们已经看到 find_vm_intersection()对冲突的检查，可是不知读者是否注意到，实际上检查的只是新区间的高端，对于其低端的冲突则并未检查。例如，老的边界是否恰好是一个已映射区间的终点呢？如果不是，那就说明在低端有了冲突。不过，对于低端的冲突是允许的，解决的方法是以新的映射为准。先通过 do_munmap()把原有的映射解除（见 803 行），再来建立新的映射。读者大概要问了，为什么对新区间的高端和低端有如此不同的容忍度和对待呢？读者最好先想一想，然后再往下看。

以前说过，用户空间的顶端是进程的用户空间堆栈。不管什么进程，在那里总是有一个已映射区间存在着的，所以 find_vma_intersection()中的 find_vma()其实不会返回 0，因为至少用于堆栈的那个区间总是存在的。当然，在堆栈以下也可能还有通过 mmap()或 ioremap()建立的映射区间。所以，如果新区间的高端有冲突，那就可能是与堆栈的冲突，而低端的冲突则只能是与数据段的冲突。所以，对于低端可以让进程自己对可能的错误负责，对于堆栈可就不能采取把原有的映射解除，另行建立新的映射这样的方法了。

建立新的映射时，先看看是否可以跟原有的区间合并，即通过扩展原有区间来覆盖新增的区间（826～831 行）。如果不行就得另行建立一个区间（838～852 行）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

最后，通过 make_page_present()，为新增的区间建立起对内存页面的映射。其代码见 mm/memory.c：

==================== mm/memory.c 1210 1229 ====================
[sys_brk()>do_brk()>make_page_present()]

```
/*
 * Simplistic page force-in..
 */
int make_pages_present(unsigned long addr, unsigned long end)
{
	int write;
	struct mm_struct *mm = current->mm;
	struct vm_area_struct * vma;

	vma = find_vma(mm, addr);
	write = (vma->vm_flags & VM_WRITE) != 0;
	if (addr >= end)
		BUG();
	do {
		if (handle_mm_fault(mm, vma, addr, write) < 0)
			return -1;
		addr += PAGE_SIZE;
	} while (addr < end);
	return 0;
}
```

这里所用的方法很有趣，那就是对新区间中的每一个页面模拟一次缺页异常。读者不妨想想，当从 do_brk()返回，进而从 sys_brk()返回之时，这些页面表项的映射是怎样的？如果进程从新分配的区间中读，读出的内容该是什么？往里面写，情况又会怎样？

2.13　系统调用 mmap()

一个进程可以通过系统调用 mmap()，将一个已打开文件的内容映射到它的用户空间，其用户界面为：

```
mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)
```

参数 fd 代表着一个已打开文件，offset 为文件中的起点，而 start 为映射到用户空间中的起始地址，length 则为长度。还有两个参数 prot 和 flags，前者用于对所映射区间的访问模式，如可写、可执行等等；后者则用于其它控制目的。从应用程序设计的角度来说，比之常规的文件操作，如 read()、write()、lseek()等等，将文件映射到用户空间后像访问内存一样地访问文件显然要方便得多（读者不妨设想一下对数据库文件的访问）。

在阅读本节之前，读者应先看一下前一节 sys_brk()的代码和有关说明，并且在阅读的过程中注意与 sys_brk()互相参照比较。有些内容可能要到阅读了后面几章以后，特别是“文件系统”，以后，再回过来阅读才能弄懂。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在 2.4.0 版的内核中实现这个调用的函数为 sys_mmap2()，但是在老一些的版本中另有一个函数 old_mmap()，这两个函数对应着不同的系统调用号。为保持对老版本的兼容，2.4.0 版中仍保留老的系统调用号和 old_mmap()，由不同版本的 C 语言库程序决定采用哪一个系统调用号。二者的代码都在 arch/i386/kernel/sys_i386.c 中：

==================== arch/i386/kernel/sys_i386.c 68 73 ====================

```
asmlinkage long sys_mmap2(unsigned long addr, unsigned long len,
    unsigned long prot, unsigned long flags,
    unsigned long fd, unsigned long pgoff)
{
    return do_mmap2(addr, len, prot, flags, fd, pgoff);
}
```

==================== arch/i386/kernel/sys_i386.c 91 106 ====================

```
asmlinkage int old_mmap(struct mmap_arg_struct *arg)
{
    struct mmap_arg_struct a;
    int err = -EFAULT;

    if (copy_from_user(&a, arg, sizeof(a)))
        goto out;

    err = -EINVAL;
    if (a.offset & ~PAGE_MASK)
        goto out;

    err = do_mmap2(a.addr, a.len, a.prot, a.flags, a.fd, a.offset >> PAGE_SHIFT);
out:
    return err;
}
```

可见，二者的区别仅在于传递参数的方式，它们的主体都是 do_mmap2()，其代码在同一文件中：

==================== arch/i386/kernel/sys_i386.c 42 66 ====================

[sys_mmap2()>do_mmap2()]

```
/* common code for old and new mmaps */
static inline long do_mmap2(
    unsigned long addr, unsigned long len,
    unsigned long prot, unsigned long flags,
    unsigned long fd, unsigned long pgoff)
{
    int error = -EBADF;
    struct file * file = NULL;

    flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
    if (!(flags & MAP_ANONYMOUS)) {
        file = fget(fd);
        if (!file)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                goto out;
        }

        down(&current->mm->mmap_sem);
        error = do_mmap_pgoff(file, addr, len, prot, flags, pgoff);
        up(&current->mm->mmap_sem);

        if (file)
            fput(file);
out:
        return error;
}
```

一般而言，系统调用 mmap()将已打开文件映射到用户空间。但是有个例外，那就是可以在调用参数 flags 中把标志位 MAP_ANONYMOUS 设成 1，表示没有文件，实际上只是用来“圈地”，即在指定的位置上分配空间。除此之外，操作的主体就是 do_mmap_pgoff()。

内核中还有个 inline 函数 do_mmap()，是供内核自己用的，它也是将已打开文件映射到当前进程的用户空间。以后，在阅读系统调用 sys_execve()的代码时，在函数 load_aout_binary()中可以看到通过 do_mmap()将可执行程序（二进制代码）映射到当前进程的用户空间。此外，do_mmap()还用来创建作为进程间通信手段的“共享内存区”。这个 inline 函数是在 include/linux/mm.h 中定义的：

==================== include/linux/mm.h 428 439 ====================

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
        unsigned long len, unsigned long prot,
        unsigned long flag, unsigned long offset)
{
        unsigned long ret = -EINVAL;
        if ((offset + PAGE_ALIGN(len)) < offset)
            goto out;
        if (!(offset & ~PAGE_MASK))
            ret = do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
out:
        return ret;
}
```

与 do_mmap2()作一比较，就可发现二者基本上相同，都是通过 do_mmap_pgoff()完成操作。不同的只是 do_mmap()不支持 MAP_ANONYMOUS；另一方面由于在进入 do_mmap()之前已经在临界区内，所以也不再需要通过信号量操作 down()和 up()加以保护。

函数 do_mmap_pgoff()的代码在 mm/mmap.c 中：

==================== mm/mmap.c 188 249 ====================

[sys_mmap2()>do_mmap2()>do_mmap_pgoff()]

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
        unsigned long prot, unsigned long flags, unsigned long pgoff)
{
        struct mm_struct * mm = current->mm;
        struct vm_area_struct * vma;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	int correct_wcount = 0;
	int error;

	if (file && (!file->f_op || !file->f_op->mmap))
		return -ENODEV;

	if ((len = PAGE_ALIGN(len)) == 0)
		return addr;

	if (len > TASK_SIZE || addr > TASK_SIZE-len)
		return -EINVAL;

	/* offset overflow? */
	if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
		return -EINVAL;

	/* Too many mappings? */
	if (mm->map_count > MAX_MAP_COUNT)
		return -ENOMEM;

	/* mlock MCL_FUTURE? */
	if (mm->def_flags & VM_LOCKED) {
		unsigned long locked = mm->locked_vm << PAGE_SHIFT;
		locked += len;
		if (locked > current->rlim[RLIMIT_MEMLOCK].rlim_cur)
			return -EAGAIN;
	}

	/* Do simple checking here so the lower-level routines won't have
	 * to. we assume access permissions have been handled by the open
	 * of the memory object, so we don't do any here.
	 */
	if (file != NULL) {
		switch (flags & MAP_TYPE) {
		case MAP_SHARED:
			if ((prot & PROT_WRITE) && !(file->f_mode & FMODE_WRITE))
				return -EACCES;

			/* Make sure we don't allow writing to an append-only file.. */
			if (IS_APPEND(file->f_dentry->d_inode) && (file->f_mode & FMODE_WRITE))
				return -EACCES;

			/* make sure there are no mandatory locks on the file. */
			if (locks_verify_locked(file->f_dentry->d_inode))
				return -EAGAIN;

			/* fall through */
		case MAP_PRIVATE:
			if (!(file->f_mode & FMODE_READ))
```

```
                    return -EACCES;
                break;

            default:
                return -EINVAL;
            }
        }

```

首先对文件和区间两方面都作一些检查，包括起始地址与长度、已经映射的次数等等。指针 file 非 0 表示映射的是具体的文件（而不是 MAP_ANONYMOUS），所以相应 file 结构中的指针 f_op 必须指向一个 file_operations 数据结构，其中的函数指针 mmap 又必须指向具体文件系统所提供的 mmap 操作（详见第 5 章“文件系统”）。从某种意义上说，do_mmap()和 do_mmap2()提供的只是一个高层的框架；低层的文件操作是由具体的文件系统提供的。

此外，还要对文件和区间的访问权限进行检查，二者必须相符。读者可以在阅读了第 5 章以后回过来仔细看这些代码。这里我们继续往下看：

==================== mm/mmap.c 250 261 ====================

[sys_mmap2()>do_mmap2()>do_mmap_pgoff()]

```
        /* Obtain the address to map to. we verify (or select) it and ensure
         * that it represents a valid section of the address space.
         */
        if (flags & MAP_FIXED) {
            if (addr & ~PAGE_MASK)
                return -EINVAL;
        } else {
            addr = get_unmapped_area(addr, len);
            if (!addr)
                return -ENOMEM;
        }

```

调用 do_mmap_pgoff()时的参数基本上就是系统调用 mmap()的参数，如果参数 flags 中的标志位 MAP_FIXED 为 0，就表示指定的映射地址只是个参考值，不能满足时可以由内核给分配一个。所以，就通过 get_unmapped_area()在当前进程的用户空间中分配一个起始地址。其代码在 mm/mmap.c 中：

==================== mm/mmap.c 374 398 ====================

[sys_mmap2()>do_mmap2()>do_mmap_pgoff()>get_unmapped_area()]

```
/* Get an address range which is currently unmapped.
 * For mmap() without MAP_FIXED and shmat() with addr=0.
 * Return value 0 means ENOMEM.
 */
#ifndef HAVE_ARCH_UNMAPPED_AREA
unsigned long get_unmapped_area(unsigned long addr, unsigned long len)
{
    struct vm_area_struct * vmm;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (len > TASK_SIZE)
        return 0;
    if (!addr)
        addr = TASK_UNMAPPED_BASE;
    addr = PAGE_ALIGN(addr);

    for (vmm = find_vma(current->mm, addr); ; vmm = vmm->vm_next) {
        /* At this point:  (!vmm || addr < vmm->vm_end). */
        if (TASK_SIZE - len < addr)
            return 0;
        if (!vmm || addr + len <= vmm->vm_start)
            return addr;
        addr = vmm->vm_end;
    }
}
#endif
```

读者自行阅读这段程序应该不会有困难。常数 TASK_UNMAPPED_BASE 是在 include/asm-i386/processor.h 中定义的：

==================== include/asm-i386/processor.h 263 266 ====================

```
/* This decides where the kernel will search for a free chunk of vm
 * space during mmap's.
 */
#define TASK_UNMAPPED_BASE  (TASK_SIZE / 3)
```

也就是说，当给定的目标地址为 0 时，内核从（TASK_SIZE/3）即 1GB 处开始向上在当前进程的虚存空间中寻找一块足以容纳给定长度的区间、而当给定的目标地址不为 0 时，则从给定的地址开始向上寻找。函数 find_vma()在当前进程已经映射的虚存空间中找到第一个满足 vma->vm_end 大于给定地址的区间。如果找不到这么一个区间，那就说明给定的地址尚未映射，因而可以使用。

至此，只要返回的地址非 0，addr 就已经是一个符合各种要求的虚存地址了。我们回到 do_mmap_pgoff()中继续往下看（mm/mmap.c）：

==================== mm/mmap.c 262 322 ====================

[sys_mmap2()>do_mmap2()>do_mmap_pgoff()]

```
    /* Determine the object being mapped and call the appropriate
     * specific mapper. the address has already been validated, but
     * not unmapped, but the maps are removed from the list.
     */
    vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!vma)
        return -ENOMEM;

    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = vm_flags(prot,flags) | mm->def_flags;

```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
if (file) {
    VM_ClearReadHint(vma);
    vma->vm_raend = 0;

    if (file->f_mode & FMODE_READ)
        vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (flags & MAP_SHARED) {
        vma->vm_flags |= VM_SHARED | VM_MAYSHARE;

        /* This looks strange, but when we don't have the file open
         * for writing, we can demote the shared mapping to a simpler
         * private mapping. That also takes care of a security hole
         * with ptrace() writing to a shared mapping without write
         * permissions.
         *
         * We leave the VM_MAYSHARE bit on, just to get correct output
         * from /proc/xxx/maps..
         */
        if (!(file->f_mode & FMODE_WRITE))
            vma->vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
    }
} else {
    vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
    if (flags & MAP_SHARED)
        vma->vm_flags |= VM_SHARED | VM_MAYSHARE;
}
vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = pgoff;
vma->vm_file = NULL;
vma->vm_private_data = NULL;

/* Clear old maps */
error = -ENOMEM;
if (do_munmap(mm, addr, len))
    goto free_vma;

/* Check against address space limit. */
if ((mm->total_vm << PAGE_SHIFT) + len
   > current->rlim[RLIMIT_AS].rlim_cur)
    goto free_vma;

/* Private writable mapping? Check memory availability.. */
if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
   !(flags & MAP_NORESERVE)                &&
   !vm_enough_memory(len >> PAGE_SHIFT))
    goto free_vma;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

每个逻辑区间都要有个 vm_area_struct 数据结构，所以通过 kmem_cache_alloc()为待映射的区间分配一个，并加以设置。我们不妨与前一节中 do_brk()的代码作一比较，在那里只是在新增的区间不能与已有的区间合并时，才分配了一个 vm_area_struct 数据结构，而这里却是无条件的。以前我们提到过，属性不同的区段不能共存于同一逻辑区间中，而映射到一个特定的文件也是一种属性，所以总是要为之单独建立一个逻辑区间。

如果调用 do_mmap_pgoff()时的 file 结构指针为 0，则目的仅在于创建虚存区间，或者说仅在于建立从物理空间到虚存区间的映射。而如果目的在于建立从文件到虚存区间的映射，那就要把为文件设置的访问权限考虑进去（见 275～296 行）。

注意代码中的 303 行将参数 pgoff 设置到 vm_area_struct 数据结构中的 vm_pgoff 字段。这个参数代表着所映射内容在文件中的起点。有了这个起点，发生缺页异常时就可以根据虚存地址计算出相应页面在文件中的位置。所以，当断开映射时，对于文件映射页面不需要像普通换入 / 换出页面那样在页面表项中指明其去向。另一方面，这也说明了为什么这样的区间必须是独立的。

至此，代表着我们所需虚存区间的数据结构已经创建了，只是尚未插入代表当前进程虚存空间的 mm_struct 结构中。可是，在某些条件下却还不得不将它撤销。为什么呢？这里调用了一个函数 do_munmap()。它检查目标地址在当前进程的虚存空间是否已经在使用，如果已经在使用就要将老的映射撤销。要是这个操作失败，那当然不能重复映射同一个目标地址，所以就得转移到 free_vma，把已经分配的 vm_area_struct 数据结构撤销。我们已经在前一节中读过 do_munmap()的代码。也许读者会感到奇怪，这个区间不是在前面调用 ger_unmapped_area()找到的吗？怎么会原来就已映射呢？回过头去注意看一下就可知道，那只是当调用参数 flags 中的标志位 MAP_FIXED 为 0 时，而当该标志位为 1 时则尚未对此加以检查。除此之外，还有两个情况也会导致撤销已经分配的 vm_area_struct 数据结构：一个是如果当前进程对虚存空间的使用超出了为其设置的下限；另一个是在要求建立由当前进程专用的可写区间，而物理页面的数量已经（暂时）不足。

读者也许还要问：为什么不把对所有条件的检验放在分配 vm_area_struct 数据结构之前呢？问题在于，在通过 kmem_cache_alloc()分配 vm_area_struct 数据结构的过程中，有可能会发生供这种数据结构专用的 slab 已经用完，而不得不分配更多物理页面的情形。而分配物理页面的过程，则又有可能因一时不能满足要求而只好先调度别的进程运行。这样，由于可能已经有别的进程或线程，特别是由本进程 clone()出来的线程（见第 4 章）运行过了，就不能排除这些条件已经改变的可能。所以，读者在内核中常常可以看到先分配某项资源，然后检测条件，如果条件不符再将资源释放（而不是先检测条件，后分配资源）的情景。关键就在于分配资源的过程中是否有可能发生调度，以及其它进程或线程的运行有否可能改变这些条件。以这里的第三个条件为例，如果发生调度，那就明显是可能改变的。

继续往下看 do_mmap_pgoff()的代码（mm/mmap.c）：

==================== mm/mmap.c 323 372 ====================

[sys_mmap2()>do_mmap2()>do_mmap_pgoff()]

```
323         if (file) {
324             if (vma->vm_flags & VM_DENYWRITE) {
325                 error = deny_write_access(file);
326                 if (error)
327                     goto free_vma;
328                 correct_wcount = 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
329             }
330             vma->vm_file = file;
331             get_file(file);
332             error = file->f_op->mmap(file, vma);
333             if (error)
334                     goto unmap_and_free_vma;
335     } else if (flags & MAP_SHARED) {
336             error = shmem_zero_setup(vma);
337             if (error)
338                     goto free_vma;
339     }
340 
341     /* Can addr have changed??
342      *
343      * Answer: Yes, several device drivers can do it in their
344      *         f_op->mmap method. -DaveM
345      */
346     flags = vma->vm_flags;
347     addr = vma->vm_start;
348 
349     insert_vm_struct(mm, vma);
350     if (correct_wcount)
351             atomic_inc(&file->f_dentry->d_inode->i_writecount);
352 
353     mm->total_vm += len >> PAGE_SHIFT;
354     if (flags & VM_LOCKED) {
355             mm->locked_vm += len >> PAGE_SHIFT;
356             make_pages_present(addr, addr + len);
357     }
358     return addr;
359 
360 unmap_and_free_vma:
361     if (correct_wcount)
362             atomic_inc(&file->f_dentry->d_inode->i_writecount);
363     vma->vm_file = NULL;
364     fput(file);
365     /* Undo any partial mapping done by a device driver. */
366     flush_cache_range(mm, vma->vm_start, vma->vm_end);
367     zap_page_range(mm, vma->vm_start, vma->vm_end - vma->vm_start);
368     flush_tlb_range(mm, vma->vm_start, vma->vm_end);
369 free_vma:
370     kmem_cache_free(vm_area_cachep, vma);
371     return error;
372 }
```

如果要建立的是从文件到虚存区间的映射，而在调用 do_mmap()时的参数 flags 中的 MAP_DENYWRITE 标志位为 1（这个标志位在前面 273 行引用的宏操作 vm_flags()中转换成 VM_DENYWRITE），那就表示不允许通过常规的文件操作访问该文件，所以要调用 deny_write_access()

PDF-XCHANGE Click to buy NOW! www.docu-track.com

排斥常规的文件操作，详见“文件系统”一章中的有关内容。至于 get_file()，其作用只是递增 file 结构中的共享计数。

我们在这里暂不关心为共享内存区而建立的映射，所以跳过 335～339 行，将来在讲到共享内存区时，还要回过来看 shmem_zero_setup()的代码。

每种文件系统都有个 file_operations 数据结构，其中的函数指针 mmap 提供了用来建立从该类文件到虚存区间的映射的操作。那么，具体到 Linux 的 Ext2 文件系统，这个函数是什么呢？我们来看 Ext2 文件系统的 file_operations 数据结构（fs/ext2/file.c）：

==================== fs/ext2/file.c 100 100 ====================
```
struct file_operations ext2_file_operations = {
	......
```
==================== fs/ext2/file.c 105 105 ====================
```
	mmap:		generic_file_mmap,
	......
```
==================== fs/ext2/file.c 109 109 ====================
```
};
```

当打开一个文件时，如果所打开的文件在一个 Ext2 文件系统中，内核就会将 file 结构中的指针 f_op 设置成指向这个数据结构，所以上面 332 行的 file->f_op->mmap 就指向 generic_file_mmap()。这个函数的代码在 mm/filemap.c 中：

==================== mm/filemap.c 1705 1725 ====================
[sys_mmap2()>do_mmap2()>do_mmap_pgoff()>generic_file_mmap()]
```
1705 /* This is used for a general mmap of a disk file */
1706 
1707 int generic_file_mmap(struct file * file, struct vm_area_struct * vma)
1708 {
1709 	struct vm_operations_struct * ops;
1710 	struct inode *inode = file->f_dentry->d_inode;
1711 
1712 	ops = &file_private_mmap;
1713 	if ((vma->vm_flags & VM_SHARED) && (vma->vm_flags & VM_MAYWRITE)) {
1714 		if (!inode->i_mapping->a_ops->writepage)
1715 			return -EINVAL;
1716 		ops = &file_shared_mmap;
1717 	}
1718 	if (!inode->i_sb || !S_ISREG(inode->i_mode))
1719 		return -EACCES;
1720 	if (!inode->i_mapping->a_ops->readpage)
1721 		return -ENOEXEC;
1722 	UPDATE_ATIME(inode);
1723 	vma->vm_ops = ops;
1724 	return 0;
1725 }
```

这个函数很简单，实质性的操作就是 1723 行将虚存区间控制结构中的指针 vm_ops 设置成 ops。至于 ops，则根据映射为专有或共享而分别指向数据结构 file_private_mmap 或 file_shared_mmap。这两

PDF-XCHANGE Click to buy NOW! www.docu-track.com

个结构均定义于 mm/filemap.c：

==================== mm/filemap.c 1686 1703 ====================

```
/*
 * Shared mappings need to be able to do the right thing at
 * close/unmap/sync. They will also use the private file as
 * backing-store for swapping..
 */
static struct vm_operations_struct file_shared_mmap = {
	nopage:		filemap_nopage,
};

/*
 * Private mappings just need to be able to load in the map.
 *
 * (This is actually used for shared mappings as well, if we
 * know they can't ever get write permissions..)
 */
static struct vm_operations_struct file_private_mmap = {
	nopage:		filemap_nopage,
};
```

数据结构的初始化也是 gcc 对 C 语言所作改进之一。这里表示具体 vm_operations_struct 结构中除 nopage 以外，所有成分的初始值均为 0 或 NULL，而 nopage 的初始值则为 filemap_nopage。相比之下在老版本中则必须写成{NULL, NULL, filemap_nopage），那样，一来麻烦，二来结构中各字段与其初始值的对应关系也不直观。

两个结构其实是一样的，都只是为缺页异常提供了 nopage 操作。此外，在 generic_file_mmap()中还检验了用于页面读／写的函数是否存在（见 1714 和 1720 行）。这两个函数应该由文件的 inode 数据结构间接地提供。在 inode 结构中有个指针 i_mapping，它指向一个 address_space 数据结构，读者应该回到“物理页面的使用和周转”一节中看一下它的定义。我们这里关心的是 address_space 结构中的指针 a_ops，它指向一个 address_space_operations 数据结构。不同的文件系统（页面交换设备可以看作是一种特殊的文件系统）有不同的 address_space_operations 结构。对于 Ext2 文件系统是 ext2_aops，定义于 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 669 676 ====================

```
struct address_space_operations ext2_aops = {
	readpage: ext2_readpage,
	writepage: ext2_writepage,
	sync_page: block_sync_page,
	prepare_write: ext2_prepare_write,
	commit_write: generic_commit_write,
	bmap: ext2_bmap
};
```

这个数据结构提供了用来读／写 ext2 文件页面的函数 ext2_readpage()和 ext2_writepage()。这些有关的数据结构和指针也是在打开文件时设置好了的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

完成了这些检查和处理，把新建立的 vm_area_struct 结构插入到当前进程的 mm_struct 结构中，就基本完成了 do_mmap_pgoff()的操作，仅在要求对区间加锁时才调用 make_pages_present()，建立起初始的页面映射，这个函数的代码已经在前一节中看到过了。

读者也许感到困惑，在文件与虚存区间之间建立映射难道就这么简单？而且我们根本就没有看到页面映射的建立！其实，具体的映射是非常动态、经常在变的。所谓文件与虚存区间之间的映射包含着两个环节，一是物理页面与文件映象之间的换入／换出，二是物理页面与虚存页面之间的映射。这二者都是动态的。所以，重要的并不是建立起一个特定的映射，而是建立起一套机制，使得一旦需要时就可以根据当时的具体情况建立起新的映射。另一方面，在计算机技术中有一个称为“lazy computation”的概念，就是说有些为将来作某种准备而进行的操作（计算）可能并无必要，所以应该推迟到真正需要时才进行。这是因为实际运行中的情况千变万化，有时候花了老大的劲才完成了准备，实际上却根本没有用到或者只用到了很小一部分，从而造成了浪费。就以这里的文件映射来说，也许映射了 100 个页面，而实际上在相当长的时间里只用到了其中的一个页面，而映射 99 个页面的开销却是不能忽略不计的。何况，长期不用的页面还得费劲把它们换出哩。考虑到这些因素，还不如到真正需要用到一个页面时再来建立该页面的映射，用到几个页面就映射几个页面。当然，那样很可能会因为分散处理而使具体映射每一个页面的开销增加。所以这里有个利弊权衡的问题，具体的决定往往要建立在统计数据的基础上。这里正是运用了这个概念，把具体页面的映射推迟到真正需要的时候才进行。具体地，就是为映射的建立、物理页面的换入和换出（以及映射的拆除）分别准备一些函数，这就是 filemap_nopage()、ext2_readpage()以及 ext2_writepage()。

那么，什么时候，由谁来调用这些函数呢？

(1) 首先，当这个区间中的一个页面首次受到访问时，会由于见面无映射而发生缺页异常，相应的异常处理程序为 do_no_page()。对于 Ext2 文件系统，do_no_page()会通过 ext2_readpage()分配一个空闲内存页面并从文件读入相应的页面，然后建立起映射。

(2) 建立起映射以后，对页面的写操作使页面变“脏”，但是页面的内容并不立即写回文件中，而由内核线程 bdflush()周期性运行时通过 page_launder()间接地调用 ext2_writepage()，将页面的内容写入文件。如果页面很长时间没有受到访问，则页面会耗尽它的寿命，从而在一次 try_to_swap_out()中被解除映射而转入不活跃状态。如果页面是“脏”的，则也会在 page_launder()中调用 ext2_writepage()。我们在 try_to_swap_out()的代码中曾看到，对用于文件映射的页面与普通的换入／换出页面有不同的处理。对于前者是解除页面映射，把页面表项设置成 0；而对后者是断开页面映射，使页面表项指向盘上页面。

(3) 解除了映射的页面在再次受到访问时又会发生缺页异常，仍旧因页面无映射而进入 do_no_page()，而不像换入／换出页面那样进入 do_swap_page()。

我们把这些情景留给读者作为“家庭作业”。

除 mmap()以外，Linux 内核还提供了几个与之有关的系统调用，作为对 mmap()的补充。限于篇幅，我们只把它们列出于下，有兴趣或需要的读者可自行阅读这些函数的源代码。

- munmap(void *start, size_t length)
  解除由 mmap()所建立的文件映射。
- mremap(void *old_adress, size_t old_size, size_t new_size, unsigned long flags)

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这是 Linux 所特有的，用来扩大或缩小已经映射的一块空间。

- msync(const void *start, size_t length, int flags)
  把一个打开的文件映射到进程的虚存空间并进行读写之后，可以用 msync()将从地址 start 开始的 length 个字节“冲刷”到实际的文件中，使得文件的内容与内存中的内容一致。参数 flags 中有三个标志位，分别为 MS_SYNC、MS_ASYNC 和 MS_INVALIDATE。MS_SYNC 表示冲刷立刻进行，并且系统调用应该等冲刷完成时才返回。MS_ASYNC 则表示冲刷可以异步地完成，系统调用应立即返回，内核可以在适当的时机进行冲刷。而 MS_INVALIDATE，那是为同一文件被多次（由多个进程）映射的情况而设置的，表示同一文件的其它映象应被视作无效而应加以刷新。
- mlock(cost void *addr, size_t len)
  虚存空间被映射到物理空间以后，一般而言是由内核运用 LRU 算法来决定页面的换入或换出的。但有时候某些进程因运行效率的考虑需要将某些页面“锁定”在内存中，这时候就可以用 mlock()将虚存中从 addr 开始的 len 个字节，实际上是这些字节所在的页面锁定在内存中，不允许换出。
- mprotect(const void *addr, size_t len, int prot)
  最后，mprotect()用来改变一段虚存空间的保护属性。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第3章 中断、异常和系统调用

我们假定本书的读者已经具备了计算机系统结构方面的基础知识，所以本章对中断以及异常（exception）处理的原理和机制不作深入的介绍。缺乏这方面基础的读者不妨先阅读一些微处理器方面的有关材料。不过，我们也并不要求读者对相关内容已经具备了很深入的理解。事实上，随着我们的介绍和分析，特别是随着各个情景的发展和代码的阅读，读者自会逐步地加深理解。

先简要提一下，中断有两种，一种是由 CPU 外部产生的，另一种是由 CPU 本身在执行程序的过程中产生的。

外部中断，就是通常所讲的“中断”（interrupt）。对于执行中的软件来说，这种中断的发生完全是“异步”的，根本无法预测此类中断会在什么时候发生。因此，CPU（或者软件）对外部中断的响应完全是被动的。不过，软件可以通过“关中断”指令关闭对中断的响应，把它“反映情况”的途径掐断，这样就可以眼不见心不烦了（这里不考虑“不可屏蔽中断”）。

由软件产生的中断则不同，它是由专设的指令，如 X86 中的“ INT n”，在程序中有意地产生的，所以是主动的，“同步”的。只要 CPU 执行了一条 INT 指令，就知道在开始执行下一条指令之前一定要先进入中断服务程序。这种主动的中断称为“陷阱”（trap）。

此外，还有一种与中断相似的机制称为“异常”（exception），一般也是异步的，多半由于“不小心”犯了规才发生。例如，当你在程序中发出一条除法指令 DIV，而除数为 0 时，就会发生一次异常。这多半是因为不小心，而不是故意的，所以也是被动的。当然，也不排除故意的可能性。我们在第 2 章中看到过通过页面异常扩展堆栈区间的情景，那就是故意安排的。

这样，一共就有三种类似的机制，即中断、陷阱以及异常。

但是，不管是外部产生的中断还是陷阱，或者异常，不管是无意的、被动的，还是故意的、主动的，CPU 的响应过程却基本上一致。这就是：在执行完当前指令以后，或者在执行当前指令的中途，就根据中断源所提供的“中断向量”，在内存中找到相应的服务程序入口并调用该服务程序。外部中断的向量是由软件或硬件设置好了的，陷阱的向量是在“自陷”指令中发出的（INT n 中的 n），而各种异常的向量则是 CPU 的硬件结构中预先规定好的。这样，这些不同的情况就因中断向量的不同而互相区分开来了。因此，在实践中常常将这些不同的情况作为一种统一的模式加以考虑和实现，而且常常统称为“中断”。至于系统调用，一般都是通过 INT 指令实现的，所以也与中断密切相关。

本章前一部分内容讲中断，包括中断的硬件支持、软件处理以及中断响应和服务的过程；后一部分则介绍系统调用的有关内容。

3.1 X86 CPU 对中断的硬件支持

本节不讨论严格意义上的中断响应全过程（比如说，怎样获得中断向量），而是着重讨论 CPU 在响应中断时，即在得到了中断向量以后，怎样进入相应的中断服务程序的过程。这是从操作系统的角

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

度需要关心的问题。Intel X86 CPU 支持 256 个不同的中断向量，这一点至今未变。可是，早期 X86 CPU 的中断响应机制是非常原始、非常简单的。在实地址模式中，CPU 把内存中从 0 开始的 1K 字节作为一个中断向量表。表中的每个表项占四个字节，由两个字节的段地址和两个字节的位移组成。这样构成的地址便是相应中断服务程序的入口地址。这与 16 位实地址模式中的寻址方式也是一致的。但是，在这样的机制上是不能构筑现代意义的操作系统的，即使把 16 位寻址改成 32 位寻址，即使实现了页式存储管理，也还是无济于事。原因在于，这个机制中并没有提供空间切换，或者说运行模式切换的手段。为了理解这一点，让我们来看看其它的CPU是怎么做的。读者也许知道，早期的UNIX是在PDP-11上实现的。PDP-11 的 CPU 中有一个与 X86 的 FLAGS 寄存器相类似的控制状态寄存器，称为 PSW 处理器状态字)。PSW 中有一个位段决定了 CPU 的当前运行优先级和模式（系统或用户）。在用户程序中是不能通过直接修改 PSW 来达到调高优先级的目的的。在 PDP-11 的中断向量表中，每个表项由两部分组成，一部分是相应中断服务程序的入口地址，另一部分就是当 CPU 进入中断服务程序后的 PSW。当然，中断向量表的内容只有当 CPU 处于系统模式时才能改变。当中断发生时，CPU 从向量表中将 PSW 装入其控制状态寄存器，而将中断服务程序的入口地址装入程序计数器，从而达到既转入了相应的中断服务程序，又从一种运行模式切换到另一种运行模式（或优先级别）的双重目的。至于原来的 PSW 则随中断返回地址一起被压入堆栈（用户堆栈），以便 CPU 从中断服务程序返回时能回到原来的运行模式。这样，就很自然地实现了运行状态的切换。CPU 平时处于用户状态，无论是因为外部中断还是系统调用（由软件产生的中断），或是某种异常，都会通过中断向量表进入系统状态，执行完中断服务程序后返回时便又恢复原状，回到用户状态。相比之下，我们可以清楚地看到，X86 实地址模式下的中断响应过程所缺少的就是类似于 PDP-11 对 PSW 的处理。

因此，Intel 在实现保护模式时，对 CPU 的中断响应机制作了大幅度的修改。

首先，中断向量表中的表项从单纯的入口地址改成了类似于 PSW 加入口地址并且更为复杂的描述项，称为“门”（gate），意思是当中断发生时必须先通过这些门，才能进入相应的服务程序。但是，这样的门并不光是为中断而设的，只要想切换 CPU 的运行状态，即其优先级别，例如从用户的 3 级进入系统的 0 级，就都要通过一道门。而从用户态进入系统态的途径也并不只限于中断（或异常，或陷阱），还可以通过子程序调用指令 CALL 和转移指令 JMP 来达到目的。而且，当中断发生时不但可以切换 CPU 的运行状态并转入中断服务程序，还可以安排进行一次任务切换（所谓“上下文切换”），立即切换到另一个进程。同此在操作系统中可以设立一个“中断服务进程（任务）”，每当中断发生时就切换到该进程。

按不同的用途和目的，CPU 中一共有四种门，即任务门（task gate）、中断门（interrupt gate）、陷阱门（trap gate）以及调用门（call gate）。其中除任务门外其它三种门的结构基本相同，不过调用门并不是与中断向量表相联系的。

先看任务门，其大小为 64 位，结构如图 3.1 所示。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

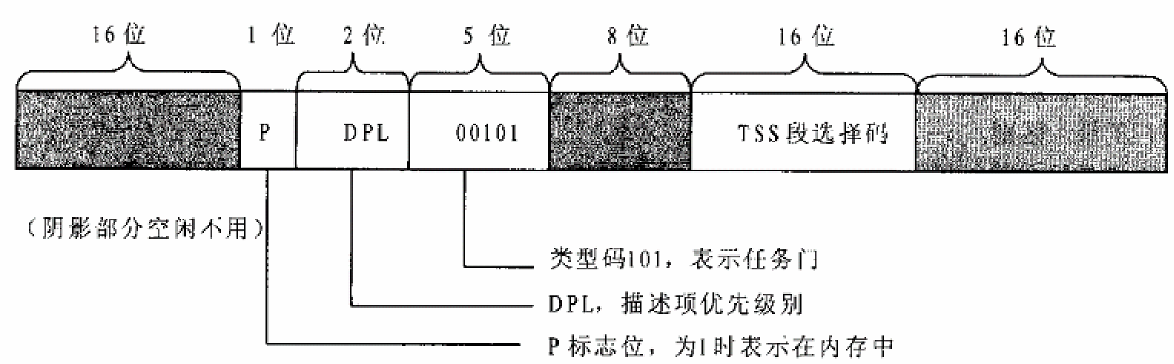


图 3.1　任务门结构图

TSS 段选择码的作用和段寄存器 CS、DS 等相似，通过 GDT 或 LDT 指向特殊的“系统段”中的一种，称为“任务状态段”（task state segment）TSS。TSS 实际上是一个用来保存任务运行“现场”的数据结构，其中包括 CPU 中所有与具体进程有关的寄存器的内容（包含页面目录指针 CR3），还包括了三个堆栈指针。中断发生时，CPU 在中断向量表中找到相应的表项。如果此表项是一个任务门，并且通过了优先级别的检查，CPU 就会将当前任务的运行现场保存在相应的 TSS 中，并将任务门所指向的 TSS 作为当前任务，将其内容装入 CPU 中的各个寄存器，从而完成了一次任务的切换。为此目的，CPU 中又增设了一个“任务寄存器 ” TR，用来指向当前任务的 TSS。在 Linux 内核中，一个任务就是一个进程，但是进程的“控制块”，即 task_struct 结构中需要存放更多的信息。所以，从这个意义上讲，Linux 的进程又并不完全是 Intel 设计意图中的任务。读者后面就会看到，Linux 内核并不采用任务门作为进程切换的手段。通过任务门切换到一个新的任务并不是惟一的途径，例如在程序中也可以用 CALL 指令或 JMP 指令通过调用门达到同样的目的。DPL 位段的作用后面还要讨论。

除任务门外，其余三种门的结构基本相同；每个门的大小也都是 64 位，见图 3.2。

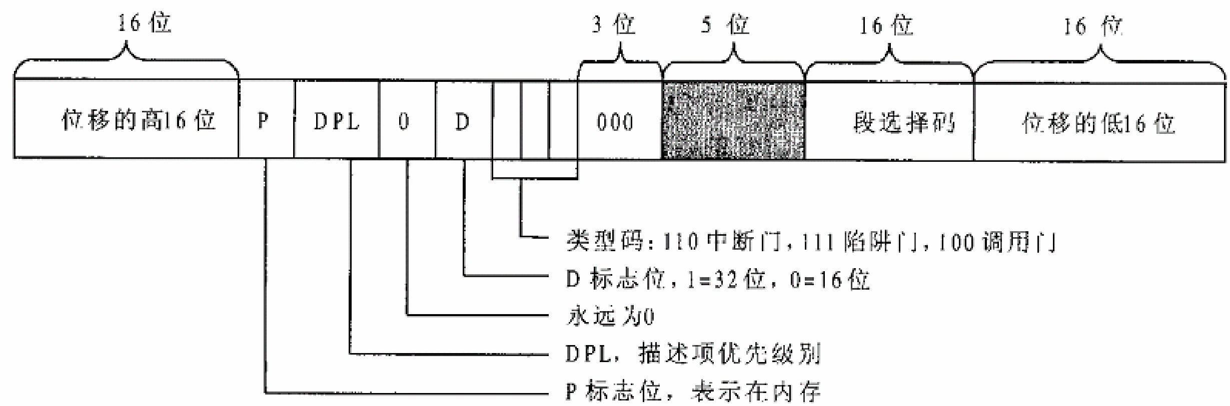


图 3.2　中断门、陷阱门和调用门结构图

三种门之间的不同之处在于 3 位的类型码。中断门的类型码是 110，陷阱门的类型码是 111，而调用门的类型码是 100。与任务门相比，不同之处主要在于：在任务门中不需要使用段内位移，因为任务门并不指向某一个子程序的入门，TSS 本身是作为一个段来对待的，而中断门、陷阱门和调用门则都

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

要指向一个子程序，所以必须结合使用段选择码和段内位移。此外，任务门中相对于 D 标志位的位置上永远是 0。

中断门和陷阱门在使用上的区别不在于中断是外部产生的或是由 CPU 本身产生的，而是在于通过中断门进入中断服务程序时 CPU 会自动将中断关闭，也就是将 CPU 中 EFLAGS 寄存器的 IF 标志位清成 0，以防嵌套中断的发生；而在通过陷阱门进入服务程序时则维持 IF 标志位不变。这就是中断门和陷阱门的惟一区别。

不管是什么门，都通过段选择码指向一个存储段。段选择码的作用与普通的段寄存器一样。我们在第 2 章中讲过，在保护模式下段寄存器的内容并不直接指向一个段的起始地址，而是指向由 GDTR 或 LDTR 决定的某个段描述表中的一个表项，所以才又称为“段选择码”。至于到底是由 GDTR 还是由 LDTR 所指向的段描述表，则取决于段选择码中的一个 TI 标志位。在 Linux 内核中，实际上只使用全局段描述表 GDT，而局部段描述表 LDT 只是在特殊应用中（主要是 WINE）才使用。对于中断门、陷阱门和调用门来说，段描述表中的相应表项显然应该是一个代码段描述项。而任务门所指向的描述项，则是专门为 TSS 而设的 TSS 描述项。TSS 描述项的结构与我们在第 2 章中所讲的基本上是相同的，但是 bit44 的 S 标志位为 0，表示不是一般的代码段或数据段。

每个段描述项中都有一个 DPL 位段，即“描述项优先级别”位段。当 CPU 通过中断门找到一个代码段描述项，并进而转入相应的服务程序时，就把这个代码段描述项装入 CPU 中，而描述项的 DPL 就变成 CPU 的当前运行级别，称为 CPL。这与我们在前面所说的 PDP-11 在中断时从向量表中同时装入 PSW 和服务程序入口地址是一致的。可是，在中断门中也有一个 DPL，那是干什么用的呢？这就是要讲到 i386 的保护模式中对运行和访问级别进行检查比对的机制了。

Intel 在 i386 CPU 中实现了一套可谓复杂得出奇的优先级别检验机制。我们在这里只根据 Linux 内核的实现介绍其中一部分。由于 Linux 内核避开了这套机制中最复杂的部分，例如不使用任务门，基本上也不使用调用门（不过为了兼容性的要求确实支持通过调用门来进入系统调用，但不是主流），再说在这里我们只关心对代码段的访问，所以剩下的部分就不太复杂了。

当通过一条 INT 指令进入一个中断服务程序时，在指令中给出一个中断向量。CPU 先根据该向量在中断向量表中找到一扇门（描述项），在这种情况下一般总是中断门。然后，就要将这个门的 DPL 与 CPU 的 CPL 相比，CPL 必须小于或等于 DPL，也就是优先级别不低于 DPL，才能穿过这扇门。不过，如果中断是由外部产生或是因 CPU 异常而产生的话，那就免去了这一层检验。穿过了中断门之后，还要进一步将目标代码段描述项中的 DPL 与 CPL 比较，目标段的 DPL 必须小于或等于 CPL。也就是说，通过中断门时只允许保持或提升 CPU 的运行级别；而不允许降低其运行级别。这两个环节中的任何一个失败都会产生一次全面保护异常（general protection exception）。

进入中断服务程序时，CPU 要将当前 EFLAGS 寄存器的内容以及返回地址压入堆栈，返回地址是由段寄存器 CS 的内容和取指令指针 EIP 的内容共同组成的。如果中断是由异常引起的，则还要将一个表示异常原因的出错代码也压入堆栈。进一步，如果中断服务程序的运行级别，也就是目标代码段的 DPL，与中断发生时的 CPL 不同，那就要引起更换堆栈。前面提到过，TSS 结构中除所有常规的寄存器内容（包括当前的 SS 和 ESP）外，还有三个额外的堆栈指针（SS 加 ESP）。这三个额外的堆栈指针分别用于当 CPU 在目标代码段中的运行级别为 0，1 以及 2 时。所以，CPU 根据寄存器 TR 的内容找到当前 TSS 结构，并根据目标代码段的 DPL，从这 TSS 结构中取出新的堆栈指针（SS 加 ESP），并装入其堆栈段寄存器 SS 和堆栈指针（寄存器）ESP，达到更换堆栈的目的。在这种情况下，CPU 不但要

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

将 EFLAGS、返回地址以及出错代码压入堆栈，还要先将原来的堆栈指针也压入堆栈（新堆栈）。示意图 3.3 也许有助于理解。

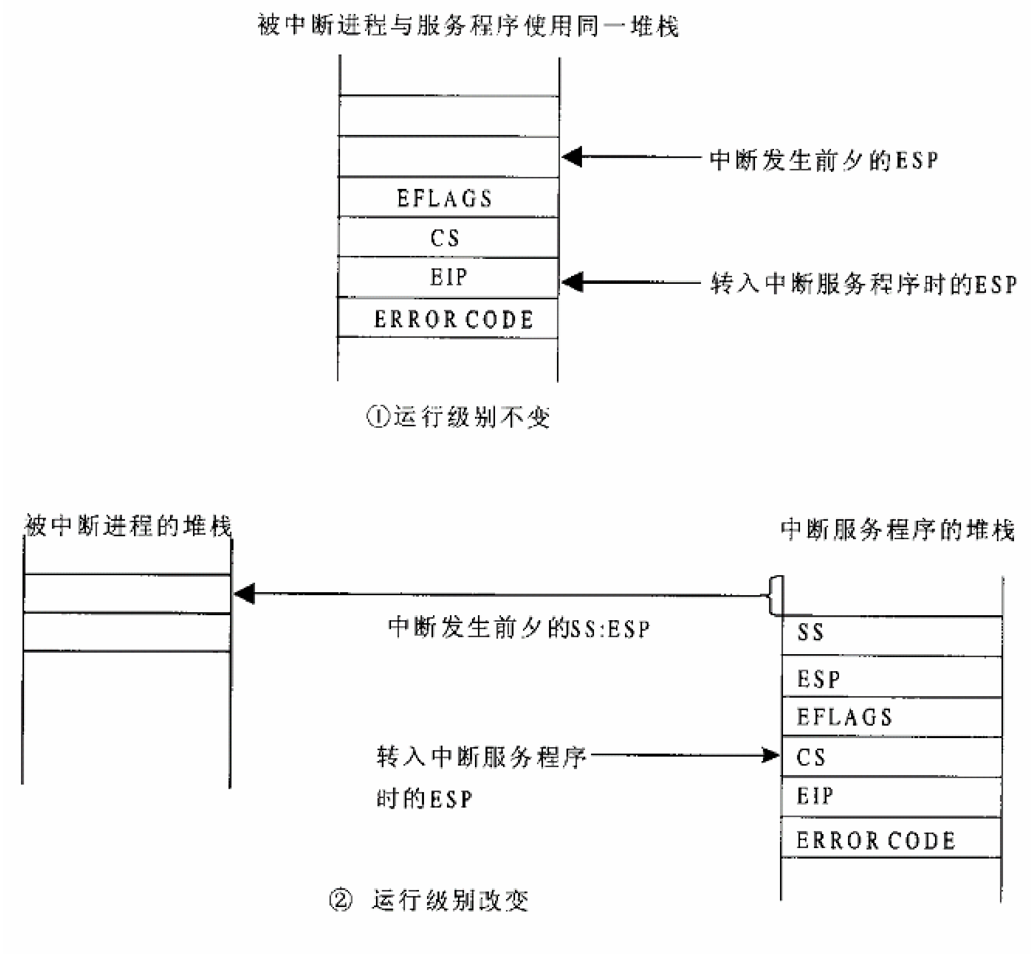


图 3.3　中断服务程序堆栈示意图

具体到 Linux 内核。当中断发生在用户状态。也就是 CPU 在用户空间中运行时，由于用户态的运行级别为 3，而在内核中的中断服务程序的运行级别为 0，所以会引起堆栈的更换。也就是说，从用户堆栈切换到系统堆栈。而当中断发生在系统状态时，也就是当 CPU 在内核中运行时，则不会更换堆栈。

最后，在保护模式中，中断向量表在内存中的位置也不再限于从地址 0 开始的地方，而是像 GDT 和 LDT 那样可以放在内存中的任何地方。为此目的，在 CPU 中又增设了一个寄存器 IDTR，指向当前中断向量表 IDT，或者说当前中断描述表。

图 3.4 的示意说明了 i386 保护模式下的中断机制在采用中断门或陷阱门时的结构。

实际的 i386 系统结构中的有关机制比上面讲的还要复杂，我们略去了其中与 Linux 内核实现无关的内容。这也从另一个角度说明，对于像 Linux 这样的操作系统（事实证明是功能最强，并且最稳定的系统之一）来说，i386 系统结构中的许多内容是不必要的，甚至是画蛇添足的，难怪有些学者批评

PDF-XCHANGE Click to buy NOW! www.docu-track.com

Intel 将 i386 的系统结构过于复杂化了。当然，也有可能将来会出现一些新的技术，从而证明 Intel 是有远见的，我们拭目以待。如果说，在能达到相同目标的前提下简单就是美，那么 i386 系统结构显然是不美的。而相比之下，Linux 内核的实现倒确实是一种美。当然，不管怎么说，i386 的系统结构能够满足像 Linux 这样的现代操作系统的需要，却是毫无疑义的。

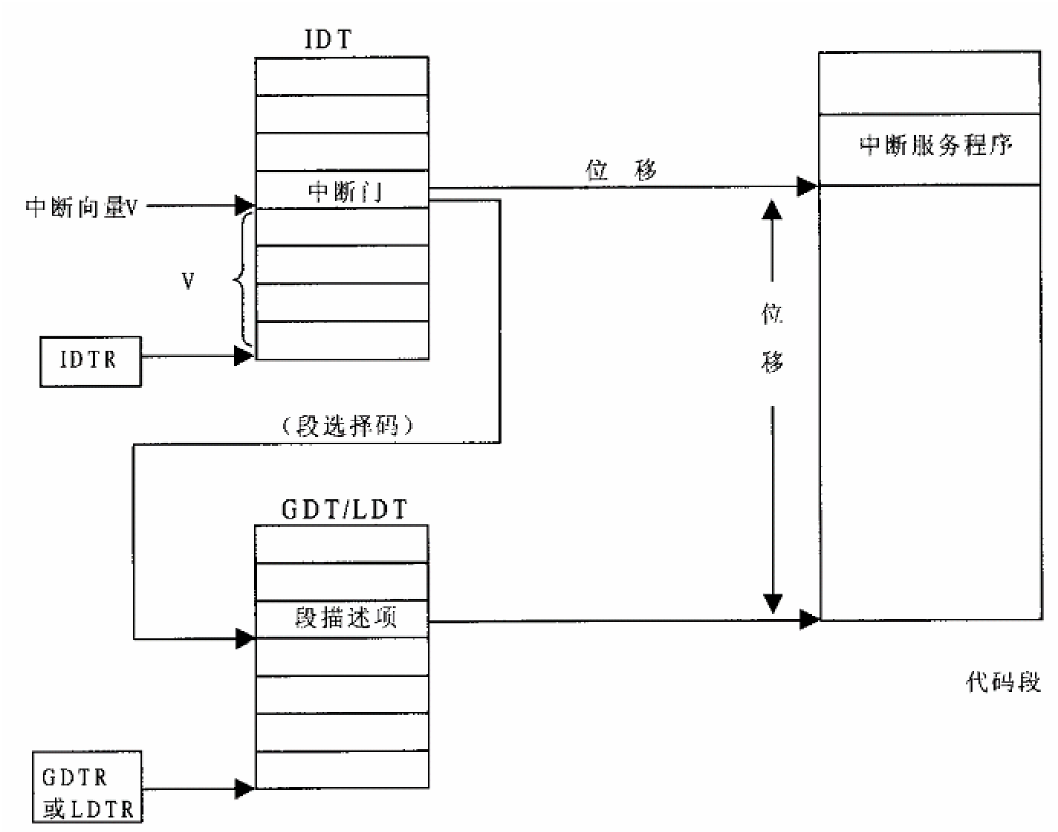


图 3.4　中断机制示意图

3.2　中断向量表 IDT 的初始化

Linux 内核在初始化阶段完成了对页式虚存管理的初始化以后，便调用 trap_init()和 init_IRQ()两个函数进行中断机制的初始化。其中 trap_init()中要是对一些系统保留的中断向量的初始化，而 init_IRQ()则主要是用于外设的中断。

函数 trap_init()是在 arch/i386/kernel/traps.c 中定义的：

==================== arch/i386/kernel/traps.c 949 996 ====================

```
void __init trap_init(void)
{
#ifdef CONFIG_EISA
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	if (isa_readl(0x0FFFD9) == 'E'+('I'<<8)+('S'<<16)+('A'<<24))
		EISA_bus = 1;
#endif

	set_trap_gate(0,&divide_error);
	set_trap_gate(1,&debug);
	set_intr_gate(2,&nmi);
	set_system_gate(3,&int3);	/* int3-5 can be called from all */
	set_system_gate(4,&overflow);
	set_system_gate(5,&bounds);
	set_trap_gate(6,&invalid_op);
	set_trap_gate(7,&device_not_available);
	set_trap_gate(8,&double_fault);
	set_trap_gate(9,&coprocessor_segment_overrun);
	set_trap_gate(10,&invalid_TSS);
	set_trap_gate(11,&segment_not_present);
	set_trap_gate(12,&stack_segment);
	set_trap_gate(13,&general_protection);
	set_trap_gate(14,&page_fault);
	set_trap_gate(15,&spurious_interrupt_bug);
	set_trap_gate(16,&coprocessor_error);
	set_trap_gate(17,&alignment_check);
	set_trap_gate(18,&machine_check);
	set_trap_gate(19,&simd_coprocessor_error);

	set_system_gate(SYSCALL_VECTOR,&system_call);

	/*
	 * default LDT is a single-entry callgate to lcall7 for iBCS
	 * and a callgate to lcall27 for Solaris/x86 binaries
	 */
	set_call_gate(&default_ldt[0],lcall7);
	set_call_gate(&default_ldt[4],lcall27);

	/*
	 * Should be a barrier for any external CPU state.
	 */
	cpu_init();

#ifdef CONFIG_X86_VISWS_APIC
	superio_init();
	lithium_init();
	cobalt_init();
#endif
}
```

程序中先设置中断向量表开头的 19 个陷阱门，这些中断向量都是 CPU 保留用于异常处理的。例如，中断向量 14 就是为页面异常保留的，CPU 硬件在页面映射及访问的过程中发生问题（如缺页），

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

就会产生一次以 14（0xe）为中断向量的异常。操作系统的设计和实现必须遵守这些规定。

然后是对系统调用向量的初始化，常数 SYSCALL_VECTOR 在 include/asm-i386/hw_irq.h 中（第 24 行）定义为 0x80，所以执行一条“ int $0x80”指令就是进行一次系统调用。

Linux 操作系统本身并不使用调用门，但是有些 Unix 变种已经用了调用门来实现系统调用，如注释中所说的 iBCS 和 Solaris/x86。为了与这些系统上编译的应用程序可执行代码相兼容，Linux 内核也相应设置了两个调用门，983 行和 984 行就是对这两个调用门的初始化。由于我们在这里并不关心 SGI 公司的特殊工作站显示设备，所以就略去了从 991 行开始的几行条件编译代码。

从程序中可以看到，这里用了三个函数来进行这些表项的初始化，那就是 set_trap_gate()、set_system_gate()以及 set_call_gate()。还有一个用于外设中断的 set_intr_gate()，这里虽然没有用到，但是也属于同一组函数（958 行调用了！！！）。这些函数都是在文件 arch/i386/kernel/traps.c 中定义的：

==================== arch/i386/kernel/traps.c 808 826 ====================

```
void set_intr_gate(unsigned int n, void *addr)
{
	_set_gate(idt_table+n,14,0,addr);
}

static void __init set_trap_gate(unsigned int n, void *addr)
{
	_set_gate(idt_table+n,15,0,addr);
}

static void __init set_system_gate(unsigned int n, void *addr)
{
	_set_gate(idt_table+n,15,3,addr);
}

static void __init set_call_gate(void *a, void *addr)
{
	_set_gate(a,12,3,addr);
}
```

这些函数都调用同一个子程序_set_gate()，设置中断描述表 idt_table 中的第 n 项，所不同的是参数表中的第 2 个、第 3 个参数。第 2 个参数对应于中断门或陷阱门格式中的 D 标志位加上类型位段。参数 14 表示 D 标志位为 1 而类型为 110，所以 set_intr_gate()设置的是中断门。第 3 个参数则对应于 DPL 位段。中断门的 DPL 一律设置成 0 是有讲究的。当中断是由外部产生或是 CPU 异常产生时，中断门的 DPL 是被忽略不顾的，所以总能穿过该中断门。可是，要是用户进程在用户空间试着用一条“ INT 2 ”来进入不可屏蔽中断的服务程序时，由于用户状态的运行级别为 3，而中断门的 DPL 为 0（级别最高），由软件产生的中断就会被拒之门外（CPU 会产生一次异常），因此不能得逞。同样，set_trap_gate()也将 DPL 设成 0，所不同的是调用_set_gate()时的第 2 个参数为 15，也即类型为 111，表示所设置的是陷阱门。我们在前面已经讲过，陷阱门与中断门的不同仅在于通过中断门进入服务程序时自动关中断，而通过陷阱门进入服务程序时则维持不变。所以，例如说，因 CPU 的页面异常而进入服务程序时，中断多半是开着的，我们在第 2 章中看到过的那些程序，如 handle_mm_fault()等等，都是可中断的。此外

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

set_system_gate()所设置的也是陷阱门，所以系统调用也是可中断的。但是 DPL 为 3，因为系统调用是在用户空间通过“ int $0x80”进行的，只有将该陷阱门的 DPL 设成 3 才能让系统调用顺利穿过，否则就会把系统调用拒之门外了。

进一步看看，这些 IDT 表项到底怎么设置。_set_gate()也在同一文件（arch/i386/kernel/traps.c）中定义：

==================== arch/i386/kernel/traps.c 788 799 ====================

```
788  #define _set_gate(gate_addr,type,dpl,addr) \
789  do { \
790    int __d0, __d1; \
791    __asm__ __volatile__ ("movw %%dx,%%ax\n\t" \
792       "movw %4,%%dx\n\t" \
793       "movl %%eax,%0\n\t" \
794       "movl %%edx,%1" \
795       :"=m" (*((long *) (gate_addr))), \
796        "=m" (*(1+(long *) (gate_addr))), "=&a" (__d0), "=&d" (__d1) \
797       :"i" ((short) (0x8000+(dpl<<13)+(type<<8))), \
798        "3" ((char *) (addr)),"2" (__KERNEL_CS << 16)); \
799  } while (0)
```

首先，do {} while (0)决定了它的循环体，也就是从 790 行至 798 行，一定会被执行一遍，并且只执行一遍。特别是在编译时不管在什么情况下都不会有问题（见第 1 章）。从 795 行的第一个“:”到 797 行的第 2 个“:”之间为输出部，其中说明了有四个变量会被改变，分别与%0、%1、%2 和%3 相结合。其中%0 与参数 gate_addr 结合，%1 与(gate_addr+1)结合，二者都是内存单元；%2 与局部变量__d0 结合，存放在寄存器%%eax 中，而%3 与局部变量__d1 结合，存放在寄存器%%edx 中。从 797 行至 798 行则为输入部。由于输出部已经定义了%0～%3，输入部中的第一个变量便为%4，而后面还有两个变量分别等价于输出部中的%3 和%2。输入部中说明的各输入变量的值，包括%3 和%2 的值，都会在引用这些变量之前设置好。

为了方便，我们把所要求的中断门（或陷阱门）的格式再表示在图 3.5。

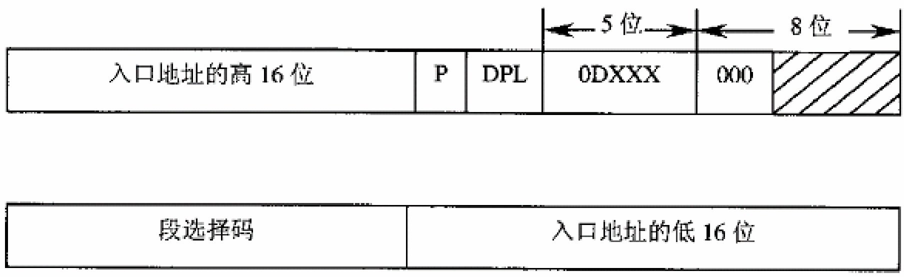


图 3.5　中断门和陷阱门的格式定义

由于 791 行要用到%%dx 和%%ax，所以编译（以及汇编）以后的代码会按输入部的说明先将%%edx 设成 addr，而%%eax 设成(__KERNEL_CS << 16)。而 791 行将%%edx 的低 16 位移入%%eax 的低 16 位（注意%%dx 与%%edx 的区别）。这样，在%%eax 中就形成了所需要的中断门的第一个长整数，其高 16 位为__KERNEL_CS，而低 16 位为 addr 的低 16 位。接着，在 792 行中将(0x8000+(dpl<<13)+(type<<8))

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

装入%%edx 的低 16 位。这样，%%edx 中高 16 位为 addr 的高 16 位，而低 16 位的 P 位为 1（因为是 0x8000），DPL 位段为 dpl（因为 dpl<<13），而 D 位加上类型位段则为 type（因为 type<<8），其余各位皆为 0。这就形成了中断门中的第 2 个长整数。然后，793 行将%%eax 写入*gate_addr，而 794 行则将%%edx 写入*(gate_addr+1)。读者不妨试试，看看能否写出效率更高的代码！当然，这种高效率是以牺牲可读性为代价的。对于像设置 IDT 表项一类并不是频繁发生的操作，这样做是否值得，大可商榷。不过，这毕竟是在内核中，而且是很底层的东西，一般也不会有很多人去读、去维护的。

系统初始化时，在 trap_init()中设置了一些为 CPU 保留专用的 IDT 表项以及系统调用所用的陷阱门以后，就要进入 init_IRQ()设置大量用于外设的通用中断门了。函数 init_IRQ()的代码在 arch/i386/kernel/i8259.c 中：

==================== arch/i386/kernel/i8259.c 438 457 ====================

```
void __init init_IRQ(void)
{
     int i;

#ifndef CONFIG_X86_VISWS_APIC
     init_ISA_irqs();
#else
     init_VISWS_APIC_irqs();
#endif
     /*
      * Cover the whole vector space, no vector can escape
      * us. (some of these will be overridden and become
      * 'special' SMP interrupts)
      */
     for (i = 0; i < NR_IRQS; i++) {
          int vector = FIRST_EXTERNAL_VECTOR + i;
          if (vector != SYSCALL_VECTOR)
               set_intr_gate(vector, interrupt[i]);
     }

```

首先是在 init_ISA_irq()中对 PC 的中断控制器 8259A 进行初始化，并且初始化一个结构数组 irq_desc[]。为什么要有这么一个结构数组呢？我们知道，i386 的系统结构支持 256 个中断向量，还要扣除一些为 CPU 本身保留的向量。但是，作为一个通用的操作系统，很难说剩下的这些中断向量是否够用。而且，很多外部设备由于种种原因可能本来就不得不共用中断向量。所以，在像 Linux 这样的系统中，限制每个中断源都必须独占使用一个中断向量是不现实的。解决的方法是为共用中断向量提供一种手段。因此，系统中为每个中断向量设置一个队列，而根据每个中断源所使用（产生）的中断向量，将其中断服务程序挂到相应的队列中去，而数组 irq_desc[]中的每个元素则是这样一个队列的头部以及控制结构。当中断发生时，首先执行与中断向量相对应的一段总服务程序，根据具体中断源的设备号在其所属队列中找到特定的服务程序加以执行。这个过程我们将在以后详细介绍，这里只要知道需要有这么一个结构数组就行了。

接着，从 FIRST_EXTERNAL_VECTOR 开始，设立 NR_IRQS 个中断向量的 IDT 表项。常数

FIRST_EXTERNAL_VECTOR 在 include/asm-i386/hw_irq.h 中（第 22 行）定义为 0x20，而 NR_IRQS 则为 224，那是在 include/asm-i386/irq.h 中（第 24 行）定义的。不过，要跳过用于系统调用的向量 0x80，那已经在前面设置好了。这里设置的服务程序入口地址都来自一个函数指针数组 interrupt[]。

函数指针数组 interrupt[]的内容也是在 arch/i386/kernel/i8259.c 中定义的：

==================== arch/i386/kernel/i8259.c 98 119 ====================

```
98   #define IRQ(x,y) \
99        IRQ##x##y##_interrupt
100
101  #define IRQLIST_16(x) \
102       IRQ(x,0), IRQ(x,1), IRQ(x,2), IRQ(x,3), \
103       IRQ(x,4), IRQ(x,5), IRQ(x,6), IRQ(x,7), \
104       IRQ(x,8), IRQ(x,9), IRQ(x,a), IRQ(x,b), \
105       IRQ(x,c), IRQ(x,d), IRQ(x,e), IRQ(x,f)
106
107  void (*interrupt[NR_IRQS])(void) = {
108       IRQLIST_16(0x0),
109
110  #ifdef CONFIG_X86_IO_APIC
111                  IRQLIST_16(0x1), IRQLIST_16(0x2), IRQLIST_16(0x3),
112       IRQLIST_16(0x4), IRQLIST_16(0x5), IRQLIST_16(0x6), IRQLIST_16(0x7),
113       IRQLIST_16(0x8), IRQLIST_16(0x9), IRQLIST_16(0xa), IRQLIST_16(0xb),
114       IRQLIST_16(0xc), IRQLIST_16(0xd)
115  #endif
116  };
117
118  #undef IRQ
119  #undef IRQLIST_16
```

数组的第一部分内容定义于 107 行，顺着 IRQLIST_16(x)和 IRQ(x,y)的定义到 98 行，可知关于函数指针的文字是由 gcc 的预处理自动产生的，因为符号##的作用是将字符串连接在一起。例如，当 108 行以参数 0x0（作为字符串）调用 IRQ_LIST_16()时，102 行中的 IRQ(x,0)就会在预处理阶段被替换成 IRQ0x00_interrupt。后面依次为 IRQ0x01_interrupt、IRQ0x02_interrupt、…，直到 IRQ0x0f_interrupt。这样，就利用 gcc 的预处理自动生成了所需的文字，而避免了枯燥繁琐的文字录入和编辑。所以，这一部分给出了 interrupt[]中的开头 16 个函数指针。对于单 CPU 系统结构，后面的指引就都是 NULL 了。如果是多处理器 SMP 结构，则后面还有 IRQ0x10 至 IRQ0xdf 等 208 个函数指针。

那么，从 IRQ0x00_interrupt 到 IRQ0x0f_interrupt 这 16 个函数本身是在哪儿定义的呢？请看 arch/i386/kernel/i8259.c 中的另外几行：

==================== arch/i386/kernel/i8259.c 38 51 ====================

```
#define BI(x,y) \
     BUILD_IRQ(x##y)

#define BUILD_16_IRQS(x) \
     BI(x,0) BI(x,1) BI(x,2) BI(x,3) \
     BI(x,4) BI(x,5) BI(x,6) BI(x,7) \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
44        BI(x,8) BI(x,9) BI(x,a) BI(x,b) \
45        BI(x,c) BI(x,d) BI(x,e) BI(x,f)
46
47   /*
48    * ISA PIC or low IO-APIC triggered (INTA-cycle or APIC) interrupts:
49    * (these are usually mapped to vectors 0x20-0x2f)
50    */
51   BUILD_16_IRQS(0x0)
```

可见，51 行的宏定义 BUILD_16_IRQS(0x0)在预处理阶段会被展开成从 BUILD_IRQ(0x00)至 BUILD_IRQ(0x0f)共 16 项宏定义的引用。而 BUILD_IRQ()则是在 include/asm-i386/hw_irq.h 中定义的：

==================== include/asm-i386/hw_irq.h 172 178 ====================

```
#define BUILD_IRQ(nr) \
asmlinkage void IRQ_NAME(nr); \
__asm__( \
"\n"__ALIGN_STR"\n" \
SYMBOL_NAME_STR(IRQ) #nr "_interrupt:\n\t" \
     "pushl $"#nr"-256\n\t" \
     "jmp common_interrupt");
```

==================== include/asm-i386/hw_irq.h 110 111 ====================

```
#define IRQ_NAME2(nr) nr##_interrupt(void)
#define IRQ_NAME(nr) IRQ_NAME2(IRQ##nr)
```

经过 gcc 的预处理以后，便会展开成一系列如下式样的代码：

```
asmlinkage void IRQ0x01_interrupt();
__asm__ ( \
"\n" \
"IRQ0x01_interrupt: \n\t" \
"pushl $0x01 - 256 \n\t" \
"jmp common_interrupt");
```

由此可以看出，实际上由外设产生的中断处理全都进入一段公共的程序 common_interrupt 中，而在此之前分别跑到 IRQ0x01_interrupt 或者 IRQ0x02_interrupt 等等的目的，只在于由此得到一个与中断向量相关的数值（压入堆栈中）对应于 IRQ0x00_interrupt 到 IRQ0x0f_interrupt，该数值分别为 0x0fffff00 至 0xffffff0f，余类推。至于 common_interrupt，那也是由 gcc 的预处理展开一个宏定义 BUILD_COMMON_IRQ()而生成的，这段程序我们在后面的情景中还要讲，这里先从略。

回到 init_IRQ()中继续往下看（arch/i386/kernel/i8259.c）：

==================== arch/i386/kernel/i8259.c 458 458 ====================

```
#ifdef CONFIG_SMP
  ......
```

==================== arch/i386/kernel/i8259.c 485 505 ====================

```
#endif

     /*
      * Set the clock to HZ Hz, we already have a valid
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         * vector now:
         */
        outb_p(0x34,0x43);      /* binary, mode 2, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);     /* LSB */
        outb(LATCH >> 8 , 0x40);    /* MSB */

#ifndef CONFIG_VISWS
        setup_irq(2, &irq2);
#endif

        /*
         * External FPU? Set up irq13 if so, for
         * original braindamaged IBM FERR coupling.
         */
        if (boot_cpu_data.hard_math && !cpu_has_fpu)
                setup_irq(13, &irq13);
}
```

由于我们在这里既不关心多处理器 SMP 结构，也不考虑 SGI 作站的特殊处理，剩下的就只是对系统时钟的初始化了。代码中有个注解，说我们已经有了个中断向量，实际上指的是 IRQ0x00_interrupt。但是要注意，虽然该中断服务的入口地址已经设置到中断向量表中，但实际上我们还没有把具体的时钟中断服务程序挂到 IRQ0 的队列中去。这个时候，这些 irq 队列都还是空的，所以即使开了中断，并且产生了时钟中断，也只不过是让它在 common_interrupt 中空跑一趟。读者以后将看到，时钟中断和对时钟中断的服务，就好像是动物的心跳、脉搏。而现在内核的脉搏尚未开始。为什么还不让它开始呢？这是因为系统在这个时候还没有完成对进程调度机制的初始化，而一旦时钟中断开始，进程调度也就要随之开始。所以，一定要等完成了对进程调度的初始化，作好了准备以后才能让脉搏开始跳动。

由此时见，设计一个真正实用的操作系统，有多少事情需要周到精细的考虑！

3.3 中断请求队列的初始化

在前一节中，我们讲到中断向量表（更准确地，应该说“中断描述表”）IDT 中有两种表项，一种是为保留专用于 CPU 本身的中断门，主要用于由 CPU 产生的异常，如“除数为 0”、“页面错”等等，以及由用户程序通过 INT 指令产生的中断（或称“陷阱”），主要用来产生系统调用（另外还有个用于 debug 的 INT 3）。这些中断门的向量除用于系统调用的 0x80 外都在 0x20 以下。从 0x20 开始就是第 2 种表项，共 224 项，都是用于外设的通用中断门。这二者的区别在于通用中断门可以为多个中断源所共享，而专用中断门则是为特定的中断源所专用。

由于通用中断门是让多个中断源共用的，而且允许这种共用的结构在系统运行的过程中动态地变化，所以在 IDT 的初始化阶段只是为每个中断向量，也即每个表项准备下一个“中断请求队列”，从而形成一个中断请求队列的数组，这就是数组 irq_desc[]。中断请求队列头部的数据结构是在 include/linux/irq.h 中定义的：

=================== include/linux/irq.h 23 55 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * Interrupt controller descriptor. This is all we need
 * to describe about the low-level hardware.
 */
struct hw_interrupt_type {
     const char * typename;
     unsigned int (*startup)(unsigned int irq);
     void (*shutdown)(unsigned int irq);
     void (*enable)(unsigned int irq);
     void (*disable)(unsigned int irq);
     void (*ack)(unsigned int irq);
     void (*end)(unsigned int irq);
     void (*set_affinity)(unsigned int irq, unsigned long mask);
};

typedef struct hw_interrupt_type  hw_irq_controller;

/*
 * This is the "IRQ descriptor", which contains various information
 * about the irq, including what kind of hardware handling it has,
 * whether it is disabled etc etc.
 *
 * Pad this out to 32 bytes for cache and indexing reasons.
 */
typedef struct {
     unsigned int status;        /* IRQ status */
     hw_irq_controller *handler;
     struct irqaction *action;   /* IRQ action list */
     unsigned int depth;         /* nested irq disables */
     spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];
```

每个队列头部中除指针 action 用来维持一个由中断服务程序描述项构成的单链队列外，还有个指针 handler 指向另一个数据结构，即 hw_interrupt_type 数据结构。那里主要是一些函数指针，用于该队列，或者说该共用“中断通道”的控制（而并不是对具体中断源的服务）。具体的函数则取决于所用的中断控制器（通常是 i8259A）。例如，函数指针 enable 和 disable 用来开启和关断其所属的通道，ack 用于对中断控制器的响应，而 end 则用于每次中断服务返回的前夕。这些函数都是在 init_IRQ()中调用 init_ISA_irqs()设置好的，见 arch/i386/kernel/i8259.c：

==================== arch/i386/kernel/i8259.c 413 436 ====================

```
void __init init_ISA_irqs (void)
{
     int i;

     init_8259A(0);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    for (i = 0; i < NR_IRQS; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = 0;
        irq_desc[i].depth = 1;

        if (i < 16) {
            /*
             * 16 old-style INTA-cycle interrupts:
             */
            irq_desc[i].handler = &i8259A_irq_type;
        } else {
            /*
             * 'high' PCI IRQs filled in on demand
             */
            irq_desc[i].handler = &no_irq_type;
        }
    }
}
```

程序先调用 init_8259A()对 8259A 中断控制器进行初始化(其代码也在 arch/i386/kernel/i8259.c 中)，然后将开头 16 个中断请求队列的 handler 指针设置成指向数据结构 i8259A_irq_type，那也是在 arch/i386/kernel/i8259.c 中定义的：

==================== arch/i386/kernel/i8259.c 148 157 ====================

```
static struct hw_interrupt_type i8259A_irq_type = {
    "XT-PIC",
    startup_8259A_irq,
    shutdown_8259A_irq,
    enable_8259A_irq,
    disable_8259A_irq,
    mask_and_ack_8259A,
    end_8259A_irq,
    NULL
};
```

用于具体中断服务程序描述项的数据结构 irqaction，则是在 include/linux/interrupt.h 中定义的：

==================== include/linux/interrupt.h 14 21 ====================

```
struct irqaction {
    void (*handler)(int, void *, struct pt_regs *);
    unsigned long flags;
    unsigned long mask;
    const char *name;
    void *dev_id;
    struct irqaction *next;
};
```

其中最主要的就是函数指针 handler，指向具体的中断服务程序。

在 IDT 表的初始化完成之初，每个中断服务队列都是空的。此时即使打开中断并且某个外设中断

PDF-XCHANGE Click to buy NOW! www.docu-track.com

真的发生了，也得不到实际的服务。虽然从中断源的硬件以及中断控制器的角度来看似乎已经得到服务了，因为形式上 CPU 确实通过中断门进入了某个中断向量的总服务程序，例如 IRQ0x01_interrupt()，并且按要求执行了对中断控制器的 ack()以及 end()，然后执行 iret 指令从中断返回。但是，从逻辑的角度、功能的角度来看，则其实并没有得到实质的服务，因为并没有执行具体的中断服务程序。所以，真正的中断服务要到具体设备的初始化程序将其中断服务程序通过 request_irq()向系统“登记”，挂入某个中断请求队列以后才会发生。

函数 request_irq()的代码在 arch/i386/kernel/irq.c 中：

==================== arch/i386/kernel/irq.c 630 705 ====================

```
/**
 *   request_irq - allocate an interrupt line
 *   @irq: Interrupt line to allocate
 *   @handler: Function to be called when the IRQ occurs
 *   @irqflags: Interrupt type flags
 *   @devname: An ascii name for the claiming device
 *   @dev_id: A cookie passed back to the handler function
 *
 *   This call allocates interrupt resources and enables the
 *   interrupt line and IRQ handling. From the point this
 *   call is made your handler function may be invoked. Since
 *   your handler function must clear any interrupt the board
 *   raises, you must take care both to initialise your hardware
 *   and to set up the interrupt handler in the right order.
 *
 *   Dev_id must be globally unique. Normally the address of the
 *   device data structure is used as the cookie. Since the handler
 *   receives this value it makes sense to use it.
 *
 *   If your interrupt is shared you must pass a non NULL dev_id
 *   as this is required when freeing the interrupt.
 *
 *   Flags:
 *
 *   SA_SHIRQ       Interrupt is shared
 *
 *   SA_INTERRUPT       Disable local interrupts while processing
 *
 *   SA_SAMPLE_RANDOM   The interrupt can be used for entropy
 *
 */

int request_irq(unsigned int irq,
         void (*handler)(int, void *, struct pt_regs *),
         unsigned long irqflags,
         const char * devname,
         void *dev_id)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int retval;
    struct irqaction * action;

#if 1
    /*
     * Sanity-check: shared interrupts should REALLY pass in
     * a real dev-ID, otherwise we'll have trouble later trying
     * to figure out which interrupt is which (messes up the
     * interrupt freeing logic etc).
     */
    if (irqflags & SA_SHIRQ) {
        if (!dev_id)
            printk("Bad boy: %s (at 0x%x) called us without a dev_id!\n", devname, (&irq)[-1]);
    }
#endif

    if (irq >= NR_IRQS)
        return -EINVAL;
    if (!handler)
        return -EINVAL;

    action = (struct irqaction *)
            kmalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->flags = irqflags;
    action->mask = 0;
    action->name = devname;
    action->next = NULL;
    action->dev_id = dev_id;

    retval = setup_irq(irq, action);
    if (retval)
        kfree(action);
    return retval;
}
```

参数 irq 为中断请求队列的序号，也就是人们通常所说的“中断请求号”，对应于中断控制器中的一个通道，有时候要在接口卡上通过微型开关或跳线来设置。但是要注意，这样的中断请求号与 CPU 所用的“中断号”或“中断向量”是不同的，中断请求号 IRQ0 相当于中断向量 0x20。也许，可以把这种中断请求号看成“逻辑”中断向量，而后者则为“物理”中断向量。通常，前 16 个断请求通道 IRQ0 至 IRQ15 是由中断控制器 i8259A 控制的。参数 irqflags 是一些标志位，其中的 SA_SHIRQ 标志表示与其它中断源公用该中断请求通道。此时必须提供一个非零的 dev_id 以供区别。当中断发生时，参数 dev_id 会被作为调用参数传回所指定的服务程序。至于这 dev_id 到底是什么，request_irq()和中断服务的总控并不在乎，只要各个具体的中断服务程序自己能够辨识和使用即可，所以这里 dev_id 的类型为

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

void*。而 request_irq()中则对此进行检查。顺便提一下，printk()产生一个出错信息。通常是写入文件/var/log/messages 或者在屏幕上显示，取决于“守护神”syslogd 和 klogd 是否已经在运行。这里有趣的是语句中的参数(&irq)[-1]。这里 irq 是第一个调用参数，所以是最后压入堆栈的，&irq 就是参数 irq 在堆栈中的位置。那么，在&irq 下面的是什么呢？那就是函数的返回地址。所以，这个 printk()语句显示该 request_irq()函数是从什么地方调用的，使程序员可以根据这个地址发现是在哪个函数中调用的。

在分配并设置了一个 irqaction 数据结构 action 以后，便调用 setup_irq()，将其链入相应的中断请求队列。其代码在同一文件（arch/i386/kernel/irq.c）中：

==================== arch/i386/kernel/irq.c 958 1014 ====================

```
/* this was setup_x86_irq but it seems pretty generic */
int setup_irq(unsigned int irq, struct irqaction * new)
{
     int shared = 0;
     unsigned long flags;
     struct irqaction *old, **p;
     irq_desc_t *desc = irq_desc + irq;

     /*
      * Some drivers like serial.c use request_irq() heavily,
      * so we have to be careful not to interfere with a
      * running system.
      */
     if (new->flags & SA_SAMPLE_RANDOM) {
          /*
           * This function might sleep, we want to call it first,
           * outside of the atomic block.
           * Yes, this might clear the entropy pool if the wrong
           * driver is attempted to be loaded, without actually
           * installing a new handler, but is this really a problem,
           * only the sysadmin is able to do this.
           */
          rand_initialize_irq(irq);
     }

     /*
      * The following block of code has to be executed atomically
      */
     spin_lock_irqsave(&desc->lock,flags);
     p = &desc->action;
     if ((old = *p) != NULL) {
          /* Can't share interrupts unless both agree to */
          if (!(old->flags & new->flags & SA_SHIRQ)) {
               spin_unlock_irqrestore(&desc->lock,flags);
               return -EBUSY;
          }

          /* add new interrupt at end of irq queue */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
996             do {
997                  p = &old->next;
998                  old = *p;
999             } while (old);
1000            shared = 1;
1001       }
1002
1003       *p = new;
1004
1005       if (!shared) {
1006            desc->depth = 0;
1007            desc->status &= ~(IRQ_DISABLED | IRQ_AUTODETECT | IRQ_WAITING);
1008            desc->handler->startup(irq);
1009       }
1010       spin_unlock_irqrestore(&desc->lock,flags);
1011
1012       register_irq_proc(irq);
1013       return 0;
1014 }
```

计算机系统在使用中常常有产生随机数的要求，但是要产生真正的随机数是不可能的（所以由计算机产生的随机数称为“伪随机数”）。为了达到尽可能的随机，需要在系统的运行中引入一些随机的因素，称为“熵”（entropy）。由各种中断源产生的中断请求在时间上大多是相当随机的，可以用来作为这样的随机因素。所以 Linux 内核提供了一种手段，使得可以根据中断发生的时间来引入一点随机性。需要在某个中断请求队列，或者说中断请求通道中引入这种随机性时，可以在调用参数 irqflags 中将标志位 SA_SAMPLE_RANDOM 设成 1。而这里调用的 rand_initialize_irq()就据此为该中断请求队列初始化一个数据结构，用来记录该中断的时序。

可想而知，对于中断请求队列的操作当然不允许受到干扰，必须要在临界区内进行，不光中断要关闭，还要防止可能来自其它处理器的干扰。代码中 986 行的 spin_lock_irqsave()就使 CPU 进入了这样的临界区。我们将在本书下册“多处理器 SMP 结构”一章中介绍和讨论 spin_lock_irqsave()，与之相对的 spin_unlock_irqrestore()则是临界区的出口。

对第一个加入队列的 irqaction 结构的处理比较简单（1003 行），不过此时要对队列的头部进行一些初始化（1006～1008 行），包括调用本队列的 startup 函数。对于后来加入队列的 irqaction 结构则要稍加检查，检查的内容为是否允许共用一个中断通道，只有在新加入的结构以及队列中的第一个结构都允许共用时才将其链入队列的尾部。

在内核中，设备驱动程序一般都要通过 request_irq()向系统登记其中断服务程序。

3.4 中断的响应和服务

搞清了 i386 CPU 的中断机制和内核中有关的初始化以后，我们就可以从中断请求的发生到 CPU 的响应，再到中断服务程序的调用与返回，沿着 CPU 所经过的路线走一遍。这样，既可以弄清和理解 Linux 内核对中断响应和服务的总体的格局和安排，还可以顺着这个过程介绍内核中的一些相关的“基

础设施”。对此二者的了解和理解，有助于读者对整个内核的理解。

这里，我们假定外设的驱动程序都已经完成了初始化，并且已把相应的中断服务程序挂入到特定的中断请求队列中，系统正在用户空间正常运行（所以中断必然是开着的），并且某个外设已经产生了一次中断请求。该请求通过中断控制器 i8259A 到达了 CPU 的“中断请求”引线 INTR。由于中断是开着的，所以 CPU 在执行完当前指令后就来响应该次中断请求。

CPU 从中断控制器取得中断向量，然后根据具体的中断向量从中断向量表 IDT 中找到相应的表项，而该表项应该是一个中断门。这样，CPU 就根据中断门的设置而到达了该通道的总服务程序的入口，假定为 IRQ0x03_interrupt。由于中断是当 CPU 在用户空间中运行时发生的，当前的运行级别 CPL 为 3；而中断服务程序属于内核，其运行级别 DPL 为 0，二者不同。所以，CPU 要从寄存器 TR 所指的当前 TSS 中取出用于内核（0 级）的堆栈指针，并把堆栈切换到内核堆栈，即当前进程的系统空间堆栈。应该指出，CPU 每次使用内核堆栈时对堆栈所作的操作总是均衡的，所以每次从系统空间返回到用户空间时堆栈指针一定回到其原点，或曰“堆栈底部”。也就是说，当 CPU 从 TSS 中取出内核堆栈指针并切换到内核堆栈时，这个堆栈一定是空的。这样，当 CPU 进入 IRQ0x03_interrupt 时，堆栈中除寄存器 EFLAGS 的内容以及返回地址外就一无所有了。另外，由于所穿过的是中断门（而不是陷阱门），所以中断已被关断；在重新开启中断之前再没有其它的中断可以发生了。

中断服务的总入口 IRQ0xYY_interrupt 的代码以前已经见到过了，但为方便起见再把它列出在这里。再说，我们现在的认识也可以更深入一些了。

如前所述，所有公用中断请求的服务程序总入口是由 gcc 的预处理阶段生成的，全部都具有相同的模式：

```
__asm__ ( \
"\n" \
"IRQ0x03_interrupt: \n\t" \
"pushl $0x03 - 256 \n\t" \
"jmp common_interrupt");
```

这段程序的目的在于将一个与中断请求号相关的数值压入堆栈，使得在 common_interrupt 中可以通过这个数值来确定这次中断的来源。可是为什么要从中断请求号 0x03 中减去 256 使其变成负数呢？就用数值 0x03 不是更直截了当吗？这是因为，系统堆栈中的这个位置在因系统调用而进入内核时要用来存放系统调用号，而系统调用又与中断服务共用一部分子程序。这样，就要有个手段来加以区分。当然，要区分系统调用号和中断请求号并不非得把其中之一变成负数不可。例如，在中断请求号上加上一个常数，比方说 0x1000，也可以达到目的。但是，如果考虑到运行时的效率，那么把其中之一变成负数无疑是效率最高的。将一个整数装入到一个通用寄存器之后，要判断它是否大于等于 0 是很方便的，只要一条寄存器指令就可以了，如“ orl %%eax, %%eax”或“ testl %%ecx, %%ecx”都可以达到目的。而如果要与另一个常数相比较，那就至少要多访问一次内存。从这个例子也可以看出，内核中的有些代码看似简单，好像只是作者随意的决定，但实际上却是经过精心推敲的。

公共的跳转目标 common_interrupt()是在 include/asm-i386/hw_irq.h 中定义的：

==================== include/asm-i386/hw_irq.h 152 161 ====================
[IRQ0x03interrupt->common_interrupt]

```
152  #define BUILD_COMMON_IRQ() \
```

```
asmlinkage void call_do_IRQ(void); \
__asm__( \
    "\n" __ALIGN_STR"\n" \
    "common_interrupt:\n\t" \
    SAVE_ALL \
    "pushl $ret_from_intr\n\t" \
    SYMBOL_NAME_STR(call_do_IRQ)":\n\t" \
    "jmp "SYMBOL_NAME_STR(do_IRQ));
```

这里主要的操作是宏操作 SAVE_ALL，就是所谓“保存现场”，把中断发生前夕所有寄存器的内容都保存在堆栈中，待中断服务完毕要返回之前再来“恢复现场”。SAVE_ALL 的定义在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 86 99 ====================

```
#define SAVE_ALL \
    cld; \
    pushl %es; \
    pushl %ds; \
    pushl %eax; \
    pushl %ebp; \
    pushl %edi; \
    pushl %esi; \
    pushl %edx; \
    pushl %ecx; \
    pushl %ebx; \
    movl $(__KERNEL_DS),%edx; \
    movl %edx,%ds; \
    movl %edx,%es;
```

这里要指出两点：第一是标志位寄存器 EFLAGS 的内容并不是在 SAVE_ALL 中保存的，这是因为 CPU 在进入中断服务时已经把它的内容连同返回地址一起压入堆栈了。第二是段寄存器 DS 和 ES 原来的内容被保存在堆栈中，然后就被改成指向用于内核的__KERNEL_DS。我们在第 2 章中讲过，__KERNEL_DS 和__USER_DS 都指向从 0 开始的空间；所不同的只是运行级别 DPL 一个为 0 级，另一个为 3 级。至于原来的堆栈段寄存器 SS 和堆栈指针 SP 的内容，则或者已被压入堆栈（如果更换堆栈）；或者继续使用而无需保存（如果不更换堆栈）。这样，在 SAVE_ALL 以后，堆栈中的内容就成为图 3.6 形式。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

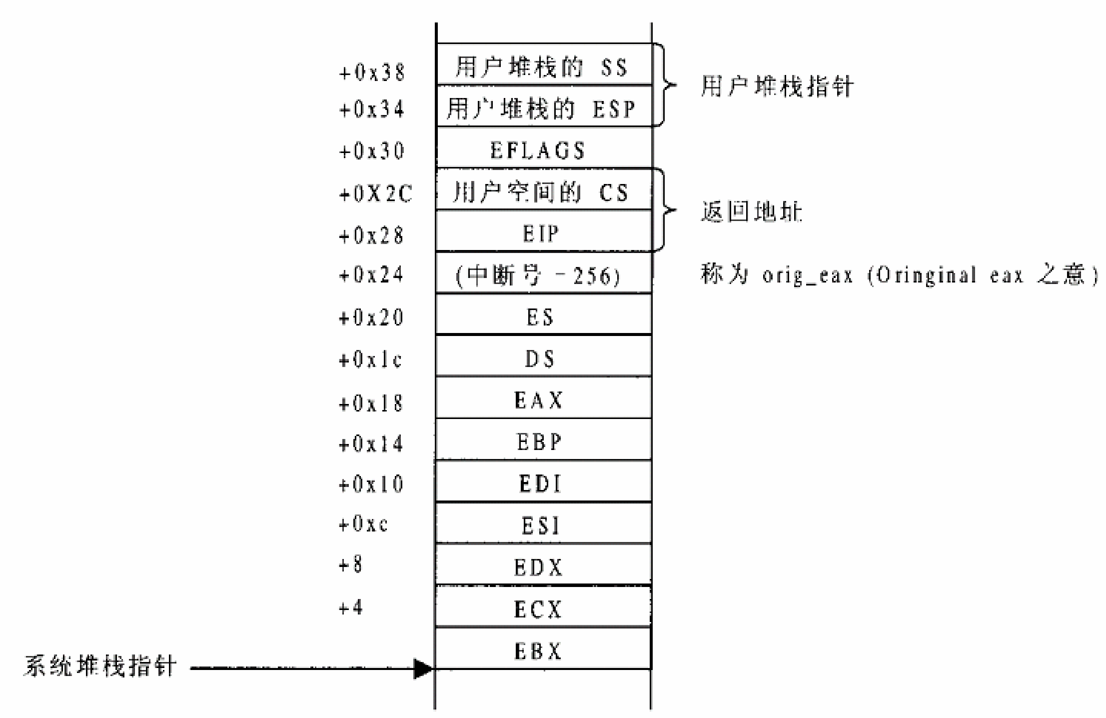


图 3.6　进入中断服务程序时系统堆栈示意图

此时系统堆栈中各项相对于堆栈指引的位置如图 3.6 所示，而 arch/i386/kernel/entry.S 中也根据这些关系定义了一些常数：

==================== arch/i386/kernel/entry.S 50 64 ====================

```
EBX       = 0x00
ECX       = 0x04
EDX       = 0x08
ESI       = 0x0C
EDI       = 0x10
EBP       = 0x14
EAX       = 0x18
DS        = 0x1C
ES        = 0x20
ORIG_EAX  = 0x24
EIP       = 0x28
CS        = 0x2C
EFLAGS         = 0x30
OLDESP         = 0x34
OLDSS          = 0x38
```

这里的 EAX，举例来说，当出现在 entry.S 的代码中时并不是表示寄存器%%eax，而是表示该寄存器的内容在系统堆栈中的位置相对于此时的堆栈指引的位移。前面在转入 common_interrupt 之前压入堆栈的（中断调用号-256）所在的位置称为 ORIG_EAX，对中断服务程序而言它代表着中断请求号。

回到 common_interrupt 的代码。在 SAVE_ALL 以后，又将一个程序标号（入口）ret_from_intr 压入堆栈，并通过 jmp 指令转入另一段程序 do_IRQ()。读者可能已注意到，IRQ0x03_interrrupt 和 common_interrupt 本质上都不是函数，它们都没有与 return 相当的指令，所以从 common_interrupt 不能返回到 IRQ0x03_interrupt，而从 IRQ0x03_interrupt 也不能执行中断返回。可是，do_IRQ()却是一个函数。所以，在通过 jmp 指令转入 do_IRQ()之前将返回地址 ret_from_intr 压入堆栈就模拟了一次函数调用，仿佛对 do_IRQ()的调用就发生在 CPU 进入 ret_from_intr 的第一条指令前夕一样。这样，当从 do_IRQ()返回时就会“返回”到 ret_from_intr 继续执行。do_IRQ()是在 arch/i386/kernel/irq.c 中定义的，我们先来看开头几行：

==================== arch/i386/kernel/irq.c 543 565 ====================

[IRQ0x03_interrupt->common_interrupt->do_IRQ()]

```
/*
 * do_IRQ handles all normal device IRQ's (the special
 * SMP cross-CPU interrupts have their own specific
 * handlers).
 */
asmlinkage unsigned int do_IRQ(struct pt_regs regs)
{
	/*
	 * We ack quickly, we don't want the irq controller
	 * thinking we're snobs just because some other CPU has
	 * disabled global interrupts (we have already done the
	 * INT_ACK cycles, it's too late to try to pretend to the
	 * controller that we aren't taking the interrupt).
	 *
	 * 0 return value means that this irq is already being
	 * handled by some other CPU. (or is disabled)
	 */
	int irq = regs.orig_eax & 0xff; /* high bits used in ret_from_ code  */
	int cpu = smp_processor_id();
	irq_desc_t *desc = irq_desc + irq;
	struct irqaction * action;
	unsigned int status;

```

函数的调用参数是一个 pt_regs 数据结构。注意，这是一个数据结构，而不是指向数据结构的指针。也就是说，在堆栈中的返回地址以上的位置上应该是一个数据结构的映象。数据结构 pt_regs 是在 include/asm-i386/ptrace.h 中定义的：

==================== include/asm-i386/ptrace.h 23 42 ====================

```
/* this struct defines the way the registers are stored on the
   stack during a system call. */

struct pt_regs {
	long ebx;
	long ecx;
	long edx;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

相信读者一定会联想到前面讲过的系统堆栈的内容并已恍然大悟：原来前面所做的一切，包括 CPU 在进入中断时自动做的，实际上都是在为 do_IRQ()建立一个模拟的子程序调用环境，使得在 do_IRQ()中既可以方便地知道进入中断前夕各个寄存器的内容，又可以在执行完毕后返回到 ret_from_intr，并且从那里执行中断返回。可想而知，当 do_IRQ()调用具体的中断服务程序时也一定会把 pt_regs 数据结构的内容传下去，不过那时只要传一个指针就够了。读者不妨回顾一下我们在第 2 章中讲过的页面异常服务程序 do_page_fault()，其调用参数表为：

```
asmlinkage void do_page_fault(struct pt_regs *regs, unsigned long error_code);
```

第一个参数就是指向 struct pt_regs 的指针，实际上就是指向系统堆栈中的那块地方。当时我们无法将这一点讲清楚，所以略了过去。而现在结合进入中断的过程一看就清楚了。不过，页面异常并不属于通用的中断请求，而是为 CPU 保留专用的，所以中断发生时并不经过 do_IRQ()这条路线，但是对于系统堆栈的这种安排基本上是一致的。

以后读者还会看到，对系统堆栈的这种安排不光用于中断，还用于系统调用。

前面讲过，在 IRQ0x03_interrupt 中把数值(0x03-256)压入堆栈的目的是使得在公共的中断处理程序中可知道中断的来源，现在进入 do_IRQ()以后的第一件事就是要弄清这一点。以 IRQ3 为例，压入堆栈的数值为 0xffffff03，现在通过 regs.orig_eax 读回来并且把高位屏蔽掉，就又得到 0x03。由于 do_IRQ()仅用于中断服务，所以不需要顾及系统调用时的情况。

代码中 561 行的 smp_processor_id()是为多处理器 SMP 结构而设的，在单处理器系统中总是返回 0。现在，既然中断请求号已经恢复，从数组 irq_desc[]中找到相应的中断请求队列当然是轻而易举的了（562 行）。下面就是对具体中断请求队列的操作了。我们继续在 do_IRQ()中往下看：

==================== arch/i386/kernel/irq.c 566 587 ====================

[IRQ0x03_interrupt->common_interrupt->do_IRQ()]

```
566         kstat.irqs[cpu][irq]++;
567         spin_lock(&desc->lock);
568         desc->handler->ack(irq);
569         /*
570            REPLAY is when Linux resends an IRQ that was dropped earlier
571            WAITING is used by probe to mark irqs that are being tested
572            */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
573        status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
574        status |= IRQ_PENDING; /* we _want_ to handle it */
575
576        /*
577         * If the IRQ is disabled for whatever reason, we cannot
578         * use the action we have.
579         */
580        action = NULL;
581        if (!(status & (IRQ_DISABLED | IRQ_INPROGRESS))) {
582             action = desc->action;
583             status &= ~IRQ_PENDING; /* we commit to handling */
584             status |= IRQ_INPROGRESS; /* we are handling it */
585        }
586        desc->status = status;
587
```

当通过中断门进入中断服务时，CPU 的中断响应机制就自动被关断了。既然已经关闭中断，为什么 567 行还要调用 spin_lock()加锁呢？这是为多处理器的情况而设置的，我们将在“多处理器 SMP 系统结构”一章中讲述，这里暂且只考虑单处理器结构。

中断处理器（如 i8259A）在将中断请求“上报”到 CPU 以后，期待 CPU 给它一个确认（ACK），表示“我已经在处理”，这里的 568 行就是做这件事。对函数指针 desc->handle->ack 的设置前面已经讲过。从 569 行至 586 行主要是对 desc->status，即中断通道状态的处理和设置，关键在于将其 IRQ_INPROGRESS 标志位设成 1，而将 IRQ_PENDING 标志位清 0。其中 IRQ_INPROGRESS 主要是为多处理器设置的，而 IRQ_PENDING 的作用则下面就会看到：

==================== arch/i386/kernel/irq.c 588 623 ====================

[IRQ0x03_interrupt->common_interrupt->do_IRQ()]

```
588        /*
589         * If there is no IRQ handler or it was disabled, exit early.
590           Since we set PENDING, if another processor is handling
591           a different instance of this same irq, the other processor
592           will take care of it.
593         */
594        if (!action)
595             goto out;
596
597        /*
598         * Edge triggered interrupts need to remember
599         * pending events.
600         * This applies to any hw interrupts that allow a second
601         * instance of the same irq to arrive while we are in do_IRQ
602         * or in the handler. But the code here only handles the _second_
603         * instance of the irq, not the third or fourth. So it is mostly
604         * useful for irq hardware that does not mask cleanly in an
605         * SMP environment.
606         */
607        for (;;) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
608             spin_unlock(&desc->lock);
609             handle_IRQ_event(irq, &regs, action);
610             spin_lock(&desc->lock);
611
612             if (!(desc->status & IRQ_PENDING))
613                 break;
614             desc->status &= ~IRQ_PENDING;
615         }
616         desc->status &= ~IRQ_INPROGRESS;
617     out:
618         /*
619          * The ->end() handler has to deal with interrupts which got
620          * disabled while the handler was running.
621          */
622         desc->handler->end(irq);
623         spin_unlock(&desc->lock);
```

如果某一个中断请求队列的服务是关闭着的（IRQ_DISABLED 标志位为 1），或者IRQ_INPROGRESS 标志位为 1，或者队列是空的，那么指针 action 为 NULL（见 580 和 582 行），无法往下执行了，所以只好返回。但是，在这几种情况下 desc->status 中的 IRQ_PENDING 标志为 1（见 574 和 583 行）。这样，以后当 CPU（在多处理器系统结构中有可能是另一个 CPU）开启该队列的服务时，会看到这个标志位而补上一次中断服务，称为“ IRQ_REPLAY”。而如果队列是空的，那么整个通道也必然是关着的，因为这是在将第一个服务程序挂入队列时才开启的。所以，这两种情形实际上相同。最后一种情况是服务已经开启，队列也不是空的，可是 IRQ_INPROGRESS 标志为 1。这只有在两种情形下才会发生。一种情形是在多处理器 SMP 系统结构中，一个 CPU 正在中断服务，而另一个 CPU 又进入了 do_IRQ()，这时候由于队列的 IRQ_INPROGRESS 标志为 1 而经 595 行返回，此时 desc->status 中的 IRQ_PENDING 标志位也是 1。第 2 种情形是在单处理器系统中 CPU 已经在中断服务程序中，但是因某种原因又将中断开启了，而且在同一个中断通道中又产生了一次中断。在这种情形下后面发生的那次中断也会因为 IRQ_INPROGRESS 标志为 1 而经 595 行返回，但也是将 desc->status 的 IN_PENDING 置成为 1。总之，这两种情形下最后的结果也是一样的，即 desc->status 中的 IRQ_PENDING 标志位为 1。

那么，IRQ_PENDING 标志位到底是怎样起作用的呢？请看 612 和 613 两行。这是在一个无限 for 循环中，具体的中断服务是在 609 行的 handle_IRQ_event()中进行的。在进入 609 行时，desc->status 中的 IRQ_PENDING 标志必然为 0。当 CPU 完成了具体的中断服务返回到 610 行以后，如果这个标志位仍然为 0，那么循环就在 613 行结束了。而如果变成了 1，那就说明已经发生过前述的某种情况，所以又循环回到 609 行再服务一次。这样，就把本来可能发生的在同一通道上（甚至可能来自同一中断源）的中断嵌套化解成为一个循环。

这样，同一个中断通道上的中断处理就得到了严格的“串行化”。也就是说，对于同一个 CPU 而言不允许中断服务嵌套，而对于不同的 CPU 则不允许并发地进入同一个中断服务程序。如果不是这样处理的话，那就要求所有的中断服务程序都必需是“可重入”的“纯代码”，那样就使中断服务程序的设计和实现复杂化了。这么一套机制的设计和实现，不能不说是非常周到、非常巧妙的。而 Linux 的稳定性和可靠性也正是植根于这种从 Unix 时代继承下来、并经过时间考验的设计中。当然，在极端的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

情况下，也有可能会发生这样的情景：中断服务程序中总是把中断打开，而中断源又不断地产生中断请求，使得 CPU 每次从 handle_IRQ_event()返回时 IRQ_PENDING 标志永远是 1，从而使 607 行的 for 循环变成一个真正的“无限”循环。如果真的发生这种情况而得不到纠正的话，那么该中断服务程序的作者应该另请高就了。

还要指出，对 desc->status 的任何改变都是在加锁的情况下进行的，这也是出于对多处理器 SMP 系统结构的考虑。

最后，在循环结束以后，只要本队列的中断服务还是开着的，就要对中断控制器执行一次“结束中断服务”操作（622 行），具体取决了中断控制器硬件的要求，所调用的函数也是在队列初始化时设置好的。

再看上面 for 循环中调用的 handle_IRQ_event()，这个函数依次执行队列中的各个中断服务程序，让它们辨认本次中断请求是否来自各自的服务对象，即中断源，如果是就进而提供相应的服务。其代码也在 arch/i386/kernel/irq.c 中：

==================== arch/i386/kernel/irq.c 418 449 ====================

[IRQ0x03_interrupt->common_interrupt->do_IRQ()>handle_IRQ_event()]

```
/*
 * This should really return information about whether
 * we should do bottom half handling etc. Right now we
 * end up _always_ checking the bottom half, which is a
 * waste of time and is not what some drivers would
 * prefer.
 */
int handle_IRQ_event(unsigned int irq, struct pt_regs * regs, struct irqaction * action)
{
    int status;
    int cpu = smp_processor_id();

    irq_enter(cpu, irq);

    status = 1;   /* Force the "do bottom halves" bit */

    if (!(action->flags & SA_INTERRUPT))
        __sti();

    do {
        status |= action->flags;
        action->handler(irq, action->dev_id, regs);
        action = action->next;
    } while (action);
    if (status & SA_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    __cli();

    irq_exit(cpu, irq);

    return status;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
449 }
```

其中 430 行的 irq_enter()和 446 行的 irq_exit()只是对一个计数器进行操作，二者均定义于 include/asm-i386/hardirq.h：

==================== include/asm-i386/hardirq.h 34 35 ====================

```
34   #define irq_enter(cpu, irq) (local_irq_count(cpu)++)
35   #define irq_exit(cpu, irq)  (local_irq_count(cpu)--)
```

当这个计数器的值为非 0 时就表示 CPU 正处于具体的中断服务程序中，以后读者会看到有些操作是不允许在此期间进行的。

一般来说，中断服务程序都是在关闭中断（不包括“不可屏蔽中断”NMI）的条件下执行的，这也是 CPU 在穿越中断门时自动关中断的原因。但是，关中断是个既不可不用，又不可滥用的手段，特别是当中断服务程序较长，操作比较复杂时，就有可能因关闭中断的时间持续太长而丢失其它的中断。经验表明，允许中断在同一个中断源或同一个中断通道嵌套是应该避免的，因此内核在 do_IRQ 中通过 IRQ_PENDING 标志位的运用来保证了这一点。可是，允许中断在不同的通道上嵌套；则只要处理得当就还是可行的。当然，必须十分小心。所以，在调用 request_irq()将一个中断服务程序挂入某个中断服务队列时，允许将参数 irqflags 中的一个标志位 SA_INTERRUPT 置成 0，表示该服务程序应该在开启中断的情况下执行。这里的 434～435 行和 444 行就是为此而设的（_sti()为开中断，_cli()为关中断）。

然后，从 437 行至 441 行的 do-while 循环就是实质性的操作了。它依次调用队列中的每一个中断服务程序。调用的参数有三个：irq 为中断请求号；action->dev_id 是一个 void 指针，由具体的服务程序自行解释和运用，这是由设备驱动程序在调用 request_irq()时自己规定的；最后一个就是前述的 pt_regs 数据结构指针 regs 了。至于具体的中断服务程序，那是设备驱动范畴内的东西，这里就不讨论了。

读者或许会问，如果中断请求队列中有多个服务程序存在，每次有来自这个通道的中断请求时就要依次把队列中所有的服务程序依次都执行一遍，岂非使效率大降？回答是：确实会有所下降，但不会严重。首先，在每个具体的中断服务程序中都应该（通常都确实是）一开始就检查各自的中断源，一般是读相应设备（接口卡上）的中断状态寄存器，看是否有来自该设备的中断请求，如没有就马上返回了，这个过程一般只需要几条机器指令；其次，每个队列中服务程序的数量一般也不会太大。所以，实际上不会有显著的影响。

最后，在 442 至 443 行，如果队列中的某个服务程序要为系统引入一些随机性的话，就调用 add_interrupt_randomness()来实现。有关详情在设备驱动一章中还会讲到。

从 handle_IRQ_event()返回的 status 的最低位必然为 1，这是在 432 行设置的。代码中还为此加了些注解（418～424 行），其作用在看了下面这一段以后就会明白。我们随着 CPU 回到 do_IRQ()中继续往下看：

==================== arch/i386/kernel/irq.c 624 628 ====================

[IRQ0x03_interrupt->common_interrupt->do_IRQ()]

```
624
625       if (softirq_active(cpu) & softirq_mask(cpu))
626           do_softirq();
627       return 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
628 }
```

到 624 行以后，从逻辑的角度说对中断请求的服务似乎已经完毕，可以返回了。可是 Linux 内核在这里有个特殊的考虑，这就是所谓 softirq，即“（在时间上）软性的中断请求”，以前称为“ bottom half”。在 Linux 中，设备驱动程序的设计人员可以将中断服务分成两“半”，其实是两“部分”，而并不一定是两“半”。第一部分是必须立即执行，一般是在关中断条件下执行的，并且必须是对每次请求都单独执行的。而另一部分，即“后半”部分，是可以稍后在开中断条件下执行的，并且往往可以将若干次中断服务中剩下来的部分合并起来执行。这些操作往往是比较费时的，因而不适宜在关中断条件下执行，或者不适宜一次占据 CPU 时间太长而影响对其它中断请求的服务。这就是所谓的“后半”（bottom half），在内核代码中常简称为 bh。作为一个比喻，读者不妨想像在“ cooked mode”下从键盘输入字符串的过程（详见设备驱动），每当按一个键的时候，首先要把字符读进来，这要放在“前半”中执行；而进一步检查所按的是否“回车”键，从而决定是否完成了一个字符串的输入，并进一步把睡眠中的进程唤醒，则可以放在“后半”中执行。

执行 bh 的机制是内核中的一项“基础设施”，所以我们在下一节单独加以介绍。这里，读者暂且只要知道有这么回事就行了。

在 do_softirq()中执行完相关的 bh 函数（如果有的话）以后，就到了从 do_IRQ()返回的时候了。返回到哪里？ entry.S 中的标号 ret_from_intr 处，这是内核中处心积虑安排好了的。其代码在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 273 280 ====================
[IRQ0x03_interrupt->common_interrupt->...->ret_from_intr]

```
273  ENTRY(ret_from_intr)
274       GET_CURRENT(%ebx)
275       movl EFLAGS(%esp),%eax      # mix EFLAGS and CS
276       movb CS(%esp),%al
277       testl $(VM_MASK | 3),%eax   # return to VM86 mode or non-supervisor?
278       jne ret_with_reschedule
279       jmp restore_all
280
```

这里的 GET_CURRENT(%ebx)将指向当前进程的 task_struct 结构的指针置入寄存器 EBX。275 行和 276 行则在寄存器 EAX 中拼凑起由中断前夕寄存器 EFLAGS 的高 16 位和代码段寄存器 CS 的（8 位）内容构成的 32 位长整数。其目的是要检验：

- 中断前夕 CPU 是否运行于 VM86 模式。
- 中断前夕 CPU 运行于用户空间还是系统空间。

VM86 模式是为在 i386 保护模式下模拟运行 DOS 软件而设置的。在寄存器 EFLAGS 的高 16 位中有个标志位表示 CPU 正在 VM86 模式中运行，我们对 VM86 模式不感兴趣，所以不予深究。而 CS 的最低两位，那就有文章了。这两位代表着中断发生时 CPU 的运行级别 CPL。我们知道 Linux 只采用两种运行级别，系统为 0，用户为 3。所以，若是 CS 的最低两位为非 0，那就说明中断发生于用户空间。

顺便说下一下 275 行的 EFLAGS(%esp)表示地址为堆栈指针%esp 的当前值加上常数 EFLAGS 处的内容，这就是保存在堆栈中的中断前夕寄存器%eflags 的内容。常数 EFLAGS 我们已经在前面介绍过，其值为 0x30。276 行中的 CS(%esp)也是一样。

如果中断发生于系统空间，控制就直接转移到 restore_all，而如果发生于用户空间（或 VM86 模式）则转移到 ret_with_reschedule。这里我们假定中断发生于用户空间，因为从 ret_with_reschedule 最终还会到达 restore_all。这段程序在同一文件（arch/i386/kernel/entry.S）中：

==================== arch/i386/kernel/entry.S 217 233 ====================
[IRQ0x03_interrupt->common_interrupt->...->ret_from_intr->ret_with_reschedule]

```
217 ret_with_reschedule:
218     cmpl $0,need_resched(%ebx)
219     jne reschedule
220     cmpl $0,sigpending(%ebx)
221     jne signal_return
222 restore_all:
223     RESTORE_ALL
224 
225     ALIGN
226 signal_return:
227     sti                 # we can get here from an interrupt handler
228     testl $(VM_MASK),EFLAGS(%esp)
229     movl %esp,%eax
230     jne v86_signal_return
231     xorl %edx,%edx
232     call SYMBOL_NAME(do_signal)
233     jmp restore_all
```

这里，首先检查是否需要进行一次进程调度。上面我们已经看到，寄存器 EBX 中的内容就是当前进程的 task_struct 结构指针，而 need_resched(%ebx)就表示该 task_struct 结构中位移为 need_resched 处的内容。220 行的 sigpending(%ebx)也是一样。常数 need_resched 和 sigpending 的定义为（见 arch/i386/kernel/entry.S）：

==================== arch/i386/kernel/entry.S 71 79 ====================

```
/*
 * these are offsets into the task-struct.
 */
state         =  0
flags         =  4
sigpending    =  8
addr_limit    = 12
exec_domain   = 16
need_resched  = 20
```

如果当前进程的 task_struct 结构中的 need_resched 字段为非 0，即表示需要进行调度，reschedule 也在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 287 289 ====================
[IRQ0x03_interrupt->common_interrupt->...->ret_from_intr->ret_with_reschedule->reschedule]

```
reschedule:
    call SYMBOL_NAME(schedule)    # test
    jmp ret_from_sys_call
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

程序在这里调用一个函数 schedule()进行调度，然后又转移到 ret_from_sys_call。我们将在系统调用一节中再加讨论。至于 schedule()则在进程一章中介绍，这里我们暂且假定不需要调度。读者以后会看到，如果要调度的话，从 ret_from_sys_call 处经过一段略为曲折的道路最终也会到达 restore_all。

同样，如果当前进程的 task_struct 结构中的 sigpending 字段为非 0，就表示该进程有“信号”等待处理，要先处理了这些待处理的信号才最后从中断返回，所以先转移到 226 行。在 228 行处先区分是否 VM86 模式，然后将寄存器%edx 的内容清 0（231 行）再调用 do_signal()。“信号（signal）”基本上是一种进程间通信的手段，我们将在“进程间通信”一章中加以介绍。处理完信号以后，控制还是回到 222 行的 restere_all。实际上，ret_from_sys_call 最后还回到 ret_from_intr，最终殊途同归都会到达 restore_all，并从那里执行中断返回。宏操作 RESTORE_ALL 的定义也在同一文件（arch/i386/kernel/entry.S）中：

==================== arch/i386/kernel/entry.S 101 112 ====================

```
101  #define RESTORE_ALL    \
102       popl %ebx;    \
103       popl %ecx;    \
104       popl %edx;    \
105       popl %esi;    \
106       popl %edi;    \
107       popl %ebp;    \
108       popl %eax;    \
109  1:   popl %ds;\
110  2:   popl %es;\
111       addl $4,%esp; \
112  3:   iret;         \
```

显然，这是与进入内核时执行的宏操作 SAVE_ALL 遥相对应的。为方便读者加以对照，我们再把 SAVE_ALL（arch/i386/kernel/entry.S）列出在这里：

==================== arch/i386/kernel/entry.S 86 100 ====================

```
#define SAVE_ALL \
     cld; \
     pushl %es; \
     pushl %ds; \
     pushl %eax; \
     pushl %ebp; \
     pushl %edi; \
     pushl %esi; \
     pushl %edx; \
     pushl %ecx; \
     pushl %ebx; \
     movl $(__KERNEL_DS),%edx; \
     movl %edx,%ds; \
     movl %edx,%es;
```

为什么在 RESTORE_ALL 的 111 行要将堆栈指针的当前值加 4？这是为了跳过 ORIG_EAX，那是

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在进入中断之初压入堆栈的中断请求号（经过变形）。我们已经看到在 do_IRQ()中的第一件事就是从中取出其最低 8 位。然后以此为下标从 irq_desc[]中找到相应的中断服务描述结构。以后在讲述系统调用和异常时读者会进一步看到其作用。读者也许会问：那为什么不像对堆栈中的其它内容一样也使用 popl 指令呢？是的，在正常的情况下确实应该使用 popl 指令，但是 pop 指令一定是与一个寄存器相联系的，现在所有的寄存器都已占满了，还能 popl 到哪儿去呢？

这样，当 CPU 到达 112 行的 iret 指令时，系统堆栈又恢复到刚进入中断门时的状态，而 iret 则使 CPU 从中断返回。跟进入中断时相对应，如果是从系统态返回到用户态就会将当前堆栈切换到用户堆栈。

3.5　软中断与 Bottom Half

中断服务一般都是在将中断请求关闭的条件下执行的，以避免嵌套而使控制复杂化。可是，如果关中断的时间持续太长就可能因为 CPU 不能及时响应其它的中断请求而使中断（请求）丢失，为此，内核允许在将具体的中断服务程序挂入中断请求队列时将 SA_INTERRUPT 标志置成 0，使这个中断服务程序在开中断的条件下执行。然而，实际的情况往往是：若在服务的全过程关中断则“扩大打击面”，而全程开中断则又造成“不安定因素”，很难取舍。一般来说，一次中断服务的过程常常可以分成两部分。开头的部分往往是必须在关中断条件下执行的。这样才能在不受干扰的条件下“原子”地完成一些关键性操作。同时，这部分操作的时间性又往往很强，必须在中断请求发生后“立即”或至少是在一定的时间限制中完成，而且相继的多次中断请求也不能合并在一起来处理。而后半部分，则通常可以、而且应该在开中断条件下执行，这样才不至于因将中断关闭过久而造成其它中断的丢失。同时，这些操作常常允许延迟到稍后才来执行，而且有可能将多次中断服务中的相关部分合并在一起处理。这些不同的性质常常使中断服务的前后两半明显地区分开来，可以、而且应该分别加以不同的实现。这里的后半部分就称为“ bottom half”，在内核代码中常常缩写为 bh。这个概念在相当程度上来自 RISC 系统结构。在 RISC 的 CPU 中，通常都有大量的寄存器。当中断发生时，要将所有这些寄存器的内容都压入堆栈，并在返回时加以恢复，为此而付出很高的代价。所以，在 RISC 结构的系统中往往把中断服务分成两部分。第一部分只保存为数不多的寄存器（内容），并利用这为数不多的寄存器来完成有限的关键性的操作，称为“轻量级中断“。而另一部分，那就相当于这里的 bh 了。虽然 i386 的结构主要是 CISC 的，面临的问题不尽相同，但前述的问题已经使 bh 的必要性在许多情况下变得很明显了。

Linux 内核为将中断服务分成两半提供了方便，并设立了相应的机制。在以前的内核中，这个机制就称为 bh。但是，在 2.4 版（确切地说是 2.3.43）中有了新的发展和推广。

以前的内核中设置了一个函数指针数组 bh_base[]，其大小为 32，数组中的每个指针可以用来指向一个具体的 bh 函数。同时，又设置了两个 32 位无符号整数 bh_active 和 bh_mask，每个无符号整数中的 32 位对应着数组 bh_base[]中的 32 个元素。

我们可以在中断与 bh 二者之间建立起一种类比。

(1) 数组 bh_base[]相当于硬件中断机制中的数组 irq_desc[]。不过 irq_desc[]中的每个元素代表着一个中断通道，所以是一个中断服务程序队列。而 bh_base[]中的每个元素却最多只能代表一个 bh 函数。但是，尽管如此，二者在概念上还是相同的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

(2) 无符号整数 bh_active 在概念上相当于硬件的“中断请求寄存器”，而 bh_mask 则相当于“中断屏蔽寄存器”。

(3) 需要执行一个 bh 函数时，就通过一个函数 mark_bh()将 bh_active 中的某一位设成 1，相当于中断源发出了中断请求，而所设置的具体标志位则类似于“中断向量”。

(4) 如果相当于“中断屏蔽寄存器”的 bh_mask 中的相应位也是 1，即系统允许执行这个 bh 函数，那么就会在每次执行完 do_IRQ()中的中断服务程序以后，以及每次系统调用结束之时，在一个函数 do_bottom_half()中执行相应的 bh 函数。而 do_bottom_half()，则类似于 do_IRQ()。

为了简化 bh 函数的设计，在 do_bottom_half()中也像 do_IRQ()中一样，把 bh 函数的执行严格地“串行化”了。这种串行化有两方面的考虑和措施：

一方面，bh 函数的执行不允许嵌套。如果在执行 bh 函数的过程中发生中断，那么由于每次中断服务以后在 do_IRQ()中都要检查和处理 bh 函数的执行，就有可能嵌套。为此，在 do_bottom_half()中针对同一 CPU 上的嵌套执行加了锁。这样，如果进入 do_bottom_half()以后发现已经上了锁，就立即返回。因为这说明 CPU 在本次中断发生之前已经在这个函数中了。

另一方面，是在多 CPU 系统中，在同一时间内最多只允许一个 CPU 执行 bh 函数，以防有两个甚至更多个 CPU 同时来执行 bh 函数而互相干扰。为此在 do_bottom_half()中针对不同 CPU 同时执行 bh 函数也加了锁。这样，如果进入 do_bottom_half()以后发现这个锁已经锁上，就说明已经有 CPU 在执行 bh 函数，所以也立即返回。

这两条措施，特别是第二条措施，保证了从单 CPU 结构到多 CPU SMP 结构的平稳过渡。可是，在当时的 Linux 内核可以在多 CPU SMP 结构上稳定运行以后，就慢慢发现这样的处理对于多 CPU SMP 结构的性能有不利的影响。原因就在于上述的第二条措施使 bh 函数的执行完全串行化了。当系统中有很多 bh 函数需要执行时，虽然系统中有多个 CPU 存在，却只有一个 CPU 这么一个“独木桥”。跟 do_IRQ()作一比较就可以发现，在 do_IRQ 中的串行化只是针对一个具体中断通道的，而 bh 函数的串行化却是全局性的，所以是“防卫过当”了。既然如此，就应该考虑放宽上述的第二条措施。但是，如果放宽了这一条，就要对 bh 函数本身的设计和实现有更高的要求（例如对使用全局量的互斥），而原来已经存在的 bh 函数显然不符合这些要求。所以，比较好的办法是保留 bh，另外再增设一种或几种机制，并把它们纳入一个统一的框架中。这就是 2.4 版中的“软中断”（softirq）机制。

从字面上说 softirq 就是软中断，可是“软中断”这个词（尤其是在中文里）已经被用作“信号”（signal）的代名词，因为信号实际上就是“以软件手段实现的中断机制”。但是，另一方面，把类似于 bh 的机制称为“软中断”又确实很贴切。这一方面反映了上述 bh 函数与中断之间的类比，另一方面也反映了这是一种在时间要求上更为软性的中断请求。实际上，这里所体现的是层次的不同。如果说“硬中断”通常是外部设备对 CPU 的中断，那么 softirq 通常是“硬中断服务程序”对内核的中断，而“信号”则是由内核（或其它进程）对某个进程的中断。后面这二者都是由软件产生的“软中断”。所以，对“软中断”这个词的含意要根据回上下文加以区分。

下面，我们以 bh 函数为主线，通过阅读代码来叙述 2.4 版内核的软中断（softirq）机制。

系统在初始化时通过函数 softirq_init()对内核的软中断机制进行初始化。其代码 kernel/softirq.c 中：

==================== kernel/softirq.c 281 290 ====================

```
281 void __init softirq_init()
282 {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int i;

    for (i=0; i<32; i++)
        tasklet_init(bh_task_vec+i, bh_action, i);

    open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
    open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

软中断本身是一种机制，同时也是一个框架。在这个框架里有 bh 机制，这是一种特殊的软中断，也可以说是设计最保守的，但却是最简单、最安全的软中断。除此之外，还有其它的软中断，定义于 include/linux/interrupt.h：

==================== include/linux/interrupt.h 56 62 ====================

```
enum
{
    HI_SOFTIRQ=0,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

这里最值得注意的是 TASKLET_SOFTIRQ，代表着一种称为 tasklet 的机制。也许采用 tasklet 这个词的原意在于表示这是一片小小的"任务"，但是这个词容易使人联想到"task"即进程而引起误会，其实这二者毫无关系。显然，NET_TX_SOFTIRQ 和 NET_RX_SOFTIRQ 两种软中断是专为网络操作而设的，所以在 softirq_init 中只对 TASKLET_SOFTIRQ 和 HI_SOFTRQ 两种软中断进行初始化。

先看 bh 机制的初始化。内核中为 bh 机制设置了一个结构数组 bh_task_vec[]，这是 tasklet_struct 数据结构的数组。

==================== kernel/softirq.c 233 233 ====================

```
struct tasklet_struct bh_task_vec[32];
```

这种数据结构的定义也在 include/linux/interrupt.h 中：

==================== include/linux/interrupt.h 97 124 ====================

```
/* Tasklets --- multithreaded analogue of BHs.

   Main feature differing them of generic softirqs: tasklet
   is running only on one CPU simultaneously.

   Main feature differing them of BHs: different tasklets
   may be run simultaneously on different CPUs.

   Properties:
   * If tasklet_schedule() is called, then tasklet is guaranteed
     to be executed on some cpu at least once after this.
   * If the tasklet is already scheduled, but its excecution is still not
     started, it will be executed only once.
```

```
 * If this tasklet is already running on another CPU (or schedule is called
   from tasklet itself), it is rescheduled for later.
 * Tasklet is strictly serialized wrt itself, but not
   wrt another tasklets. If client needs some intertask synchronization,
   he makes it with spinlocks.
 */

struct tasklet_struct
{
	struct tasklet_struct *next;
	unsigned long state;
	atomic_t count;
	void (*func)(unsigned long);
	unsigned long data;
};
```

代码的作者加了详细的注释，说 tasklet 是“多序”（不是“多进程”或“多线程”!）的 bh 函数。为什么这么说呢？因为对 tasklet 的串行化不像对 bh 函数那样严格，所以允许在不同的 CPU 上同时执行 tasklet，但必须是不同的 tasklet。一个 tasklet_struct 数据结构就代表着一个 tasklet，结构中的函数指针 func 指向其服务程序。那么，为什么在 bh 机制中要使用这种数据结构呢？这是因为 bh 函数的执行（并不是 bh 函数本身）就是作为一个 tasklet 来实现的，在此基础上再加上更严格的限制，就成了 bh。

函数 tasklet_init()的代码在 kernel/softirq.c 中：

==================== kernel/softirq.c 203 210 ====================
[softirq_init()>tasklet_init()]

```
void tasklet_init(struct tasklet_struct *t,
		  void (*func)(unsigned long), unsigned long data)
{
	t->func = func;
	t->data = data;
	t->state = 0;
	atomic_set(&t->count, 0);
}
```

在 softirq_init()中，对用于 bh 的 32 个 tasklet_struct 结构调用 tasklet_init()以后，它们的函数指针 func 全都指向 bh_action()。

对其它软中断的初始化是通过 open_softirq()完成的，其代码也在同一文件中：

==================== kernel/softirq.c 103 117 ====================
[softirq_init()>open_softirq()]

```
static spinlock_t softirq_mask_lock = SPIN_LOCK_UNLOCKED;

void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
	unsigned long flags;
	int i;

	spin_lock_irqsave(&softirq_mask_lock, flags);
```

```
    softirq_vec[nr].data = data;
    softirq_vec[nr].action = action;

    for (i=0; i<NR_CPUS; i++)
        softirq_mask(i) |= (1<<nr);
    spin_unlock_irqrestore(&softirq_mask_lock, flags);
}
```

内核中为软中断设置了一个以“软中断号”为下标的数组 softirq_vec[]，类似于中断机制中的 irq_desc[]。

==================== kernel/softirq.c 48 48 ====================

```
static struct softirq_action softirq_vec[32] __cacheline_aligned;
```

这是一个 softirq_action 数据结构的数组，其定义为：

==================== include/linux/interrupt.h 64 72 ====================

```
/* softirq mask and active fields moved to irq_cpustat_t in
 * asm/hardirq.h to get better cache usage.  KAO
 */

struct softirq_action
{
    void (*action)(struct softirq_action *);
    void *data;
};
```

数组 softirq_vec[]是个全局量，系统中的各个 CPU 所看到的是同一个数组。但是，每个 CPU 各有其自己的“软中断控制 / 状况结构”，所以这些数据结构形成一个以 CPU 编号为下标的数组 irq_stat[]。这个数组也是全局量，但是各个 CPU 可以按其自身的编号访问相应的数据结构。我们把有关的定义列出于下，供读者自己阅读：

==================== include/asm-i386/hardirq.h 8 16 ====================

```
/* entry.S is sensitive to the offsets of these fields */
typedef struct {
    unsigned int __softirq_active;
    unsigned int __softirq_mask;
    unsigned int __local_irq_count;
    unsigned int __local_bh_count;
    unsigned int __syscall_count;
    unsigned int __nmi_count;   /* arch dependent */
} ____cacheline_aligned irq_cpustat_t;
```

==================== kernel/softirq.c 45 45 ====================

```
irq_cpustat_t irq_stat[NR_CPUS];
```

==================== include/linux/irq_cpustat.h 22 30 ====================

```
#ifdef CONFIG_SMP
#define __IRQ_STAT(cpu, member)  (irq_stat[cpu].member)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#else
#define __IRQ_STAT(cpu, member)  ((void)(cpu), irq_stat[0].member)
#endif

  /* arch independent irq_stat fields */
#define softirq_active(cpu) __IRQ_STAT((cpu), __softirq_active)
#define softirq_mask(cpu)   __IRQ_STAT((cpu), __softirq_mask)
```

数据结构中的__softirq_active 相当于“软中断请求寄存器”，__softirq_mask 则相当于“软中断屏蔽寄存器”。函数 open_softirq()除把函数指针 action 填入 softirq_vec[]中的相应元素外，还把所有 CPU 的“中断屏蔽寄存器”中的相应位设置成 1，使这个软中断在每个 CPU 上都可以执行。从 softirq_init()中调用 open_softirq()把 TASKLET_SOFTIRQ 和 HI_SOFTIRQ 两种软中断的处理程序分别设置成 tasklet_action()和 tasklet_hi_action()。

内核中还有另一个以 CPU 编号为下标的数组 tasklet_hi_vec[]，这是 tasklet_head 结构数组，每个 tasklet_head 结构就是一个 tasklet_struct 结构的队列头。

==================== kernel/softirq.c 167 167 ====================

```
struct tasklet_head tasklet_hi_vec[NR_CPUS] __cacheline_aligned;
```

==================== include/linux/interrupt.h 139 142 ====================

```
struct tasklet_head
{
     struct tasklet_struct *list;
} __attribute__ ((__aligned__(SMP_CACHE_BYTES)));
```

回到 bh 机制这个话题上。通过 tasklet_init()只是使相应 tasklet_struct 结构中的函数指针 func 指向了 bh_action()，也就是建立了 bh 的执行机制，而具体的 bh 函数还没有与之挂钩，就好像具体的中断服务程序尚未挂入中断服务队列一样。具体 bh 函数是通过 init_bh()设置的。下面是取自 sched_init()中的一个片段：

==================== kernel/sched.c 1260 1262 ====================

```
     init_bh(TIMER_BH, timer_bh);
     init_bh(TQUEUE_BH, tqueue_bh);
     init_bh(IMMEDIATE_BH, immediate_bh);
```

以用于时钟中断的 bh 函数 timer_bh()为例，其“bh 向量”、或“bh 编号”为 TIMER_BH。目前内核中已经定义的编号如下：

==================== include/linux/interrupt.h 24 43 ====================

```
/* Who gets which entry in bh_base.  Things which will occur most often
   should come first */

enum {
     TIMER_BH = 0,
     TQUEUE_BH,
     DIGI_BH,
     SERIAL_BH,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        RISCOM8_BH,
        SPECIALIX_BH,
        AURORA_BH,
        ESP_BH,
        SCSI_BH,
        IMMEDIATE_BH,
        CYCLADES_BH,
        CM206_BH,
        JS_BH,
        MACSERIAL_BH,
        ISICOM_BH
};
```

再看 init_bh()的代码，这是在 kernel/softirq.c 中：

===================== kernel/softirq.c 269 273 =====================

```
void init_bh(int nr, void (*routine)(void))
{
        bh_base[nr] = routine;
        mb();
}
```

显然，这里的数组 bh_base[]就是前述的函数指针数组。这里调用的函数 mb()与 CPU 中执行指令的“流水线”有关，而这并不是我们现在所关心的。

需要执行一个特定的 bh 函数时，可以通过一个 inline 函数 mark_bh()提出请求。读者在“时钟中断”一节中可以看到在 do_timer()中通过“mark_bh(TIMER_BH);”提出对 timer_bh()的执行请求。函数 mark_bh()的代码在 include/linux/interrupt.h 中：

===================== include/linux/interrupt.h 232 235 =====================

```
static inline void mark_bh(int nr)
{
        tasklet_hi_schedule(bh_task_vec+nr);
}
```

如前所述，内核中为 bh 函数的执行设立了一个 tasklet_struct 结构数组 bh_task_vec[]，这里以 bh 函数的编号为下标就可以找到相应的数据结构，并用其调用 tasklet_hi_schedule()，其代码也在 include/linux/interrupt.h 中。读者应该还记得，在 bh_tasm_vec[]的每个 tasklet_struct 结构中，函数指针 func 都指向 bh_action()。

===================== include/linux/interrupt.h 171 183 =====================

[mark_bh()>tasklet_hi_schedule()]

```
static inline void tasklet_hi_schedule(struct tasklet_struct *t)
{
        if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state)) {
                int cpu = smp_processor_id();
                unsigned long flags;

                local_irq_save(flags);
```

```
            t->next = tasklet_hi_vec[cpu].list;
            tasklet_hi_vec[cpu].list = t;
            __cpu_raise_softirq(cpu, HI_SOFTIRQ);
            local_irq_restore(flags);
        }
}
```

这里的 smp_processor_id()返回当前进程所在 CPU 的编号，然后以此为下标从 tasklet_hi_vec[]中找到该 CPU 的队列头，把参数 t 所指的 tasklet_struct 数据结构链入这个队列。由此可见，对执行 bh 函数的要求是在哪一个 CPU 上提出的，就把它“调度”在哪一个 CPU 上执行，函数名中的“ schedule”就是这个意思，而与“进程调度”毫无关系。另一方面，一个 tasklet_struct 代表着对 bh 函数的一次执行，在同一时间内只能把它链入一个队列中，而不可能同时出现在多个队列中。对于同一个 tasklet_struct 数据结构，如果已经对其调用了 tasklet_hi_schedule()，而尚未得到执行，就不允许再将其链入队列，所以在数据结构中设置了一个标志位 TASKLET_STATE_SCHED 来保证这一点。最后，还要通过__cpu_raise_softirq()正式发出软中断请求。

==================== include/linux/interrupt.h 77 80 ====================
[mark_bh()>tasklet_hi_schedule()>__cpu_raise_softirq()]

```
static inline void __cpu_raise_softirq(int cpu, int nr)
{
        softirq_active(cpu) |= (1<<nr);
}
```

读者在前面已经看到过 softirq_active()的定义，它对给定 CPU 的“软中断控制 / 状况结构”操作，将其中__soft_irq_active 字段内的相应标志位设成 1。

内核每当在 do_IRQ()中执行完一个通道中的中断服务程序以后，以及每当从系统调用返回时，都要检查是否有软中断请求在等待执行。下面是 do_IRQ()中的一个片段：

==================== arch/i386/kernel/irq.c 625 626 ====================

```
        if (softirq_active(cpu) & softirq_mask(cpu))
            do_softirq();
```

另一段代码取自 arch/i386/kernel/entry.S，这是在从系统调用返回时执行的：

==================== arch/i386/kernel/entry.S 205 215 ====================

```
ENTRY(ret_from_sys_call)
#ifdef CONFIG_SMP
        movl processor(%ebx),%eax
        shll $CONFIG_X86_L1_CACHE_SHIFT,%eax
        movl SYMBOL_NAME(irq_stat)(,%eax),%ecx          # softirq_active
        testl SYMBOL_NAME(irq_stat)+4(,%eax),%ecx # softirq_mask
#else
        movl SYMBOL_NAME(irq_stat),%ecx      # softirq_active
        testl SYMBOL_NAME(irq_stat)+4,%ecx   # softirq_mask
#endif
        jne   handle_softirq
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== arch/i386/kernel/entry.S 282 284 ====================

```
handle_softirq:
        call SYMBOL_NAME(do_softirq)
        jmp ret_from_intr
```

注意，这里的 processor 表示 task_struct 数据结构中该字段的位移，所以 207 行是从当前进程的 task_struct 数据结构中取当前 CPU 的编号。而 SYMBOL_NAME(irq_stat)(,%eax)相当于 irq_stat[cpu]，并且是其中第一个字段；相应地，SYMBOL_NAME(irq_stat)+4(,%eax)相当这个数据结构中的第二个字段，并且第一个字段必须是 32 位。读者不妨回过去看一下 irq_cpustat_t 的定义，在那里有个注释，说 arch/i386/kernel/entry.S 中的代码对这个数据结构中的字段位置敏感，就是这个意思。所以，这些汇编代码实际上与上面 do_IRQ()中的两行 C 代码是一样的。

检测到软中断请求以后，就要通过 do_softirq()加以执行了。其代码在 kernel/softirq.c 中：

==================== kernel/softirq.c 50 100 ====================

```
asmlinkage void do_softirq()
{
        int cpu = smp_processor_id();
        __u32 active, mask;

        if (in_interrupt())
                return;

        local_bh_disable();

        local_irq_disable();
        mask = softirq_mask(cpu);
        active = softirq_active(cpu) & mask;

        if (active) {
                struct softirq_action *h;

restart:
                /* Reset active bitmask before enabling irqs */
                softirq_active(cpu) &= ~active;

                local_irq_enable();

                h = softirq_vec;
                mask &= ~active;

                do {
                        if (active & 1)
                                h->action(h);
                        h++;
                        active >>= 1;
                } while (active);

                local_irq_disable();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
84
85              active = softirq_active(cpu);
86              if ((active &= mask) != 0)
87                   goto retry;
88         }
89
90         local_bh_enable();
91
92         /* Leave with locally disabled hard irqs. It is critical to close
93          * window for infinite recursion, while we help local bh count,
94          * it protected us. Now we are defenceless.
95          */
96         return;
97
98    retry:
99         goto restart;
100   }
```

软中断服务程序既不允许在一个硬中断服务程序内部执行，也不允许在一个软中断服务程序内部执行，所以要通过一个宏操作 in_interrupt()加以检测，这是在 include/asm-i386/hardirq.h 中定义的：

==================== include/asm-i386/hardirq.h 20 25 ====================

```
/*
 * Are we in an interrupt context? Either doing bottom half
 * or hardware interrupt processing?
 */
#define in_interrupt() ({ int __cpu = smp_processor_id(); \
     (local_irq_count(__cpu) + local_bh_count(__cpu) != 0); })
```

显然，这个测试防止了软中断服务程序的嵌套，这就是前面讲的第一条串行化格施。与 local_bh_disable()有关的定义在 include/asm-i386/softirq.h 中：

==================== include/asm-i386/softirq.h 7 11 ====================

```
#define cpu_bh_disable(cpu) do { local_bh_count(cpu)++; barrier(); } while (0)
#define cpu_bh_enable(cpu)  do { barrier(); local_bh_count(cpu)--; } while (0)

#define local_bh_disable()  cpu_bh_disable(smp_processor_id())
#define local_bh_enable()   cpu_bh_enable(smp_processor_id())
```

从 do_softirq()的代码中可以看出，使 CPU 不能执行软中断服务程序的“关卡”只有一个，那就是 in_interrupt()，所以对软中断服务程序的执行并没有采取前述的第二条串行化措施。这就是说，不同的 CPU 可以同时进入对软中断服务程序的执行（见 78 行），分别执行各自所请求的软中断服务。从这个意义上，软中断服务程序的执行是“并发”的、多序的。但是，这些软中断服务程序的设计和实现必须十分小心，不能让它们互相干扰（例如通过共享的全局量）。至于 do_softirq()中其它的代码，则读者不会感到困难，我们就不多说了。

在我们这个情景中，如前所述，执行的服务程序为 bh_action()，其代码在 kernel/softirq.c 中：

==================== kernel/softirq.c 235 267 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[do_softirq()>bh_action()]

```
235 /* BHs are serialized by spinlock global_bh_lock.
236
237    It is still possible to make synchronize_bh() as
238    spin_unlock_wait(&global_bh_lock). This operation is not used
239    by kernel now, so that this lock is not made private only
240    due to wait_on_irq().
241
242    It can be removed only after auditing all the BHs.
243  */
244 spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;
245
246 static void bh_action(unsigned long nr)
247 {
248      int cpu = smp_processor_id();
249
250      if (!spin_trylock(&global_bh_lock))
251           goto resched;
252
253      if (!hardirq_trylock(cpu))
254           goto resched_unlock;
255
256      if (bh_base[nr])
257           bh_base[nr]();
258
259      hardirq_endlock(cpu);
260      spin_unlock(&global_bh_lock);
261      return;
262
263 resched_unlock:
264      spin_unlock(&global_bh_lock);
265 resched:
266      mark_bh(nr);
267 }
```

这里对具体 bh 函数的执行（见 257 行）又设置了两道关卡。一道是 hardirq_trylock()，其定义为：

==================== include/asm-i386/hardirq.h 31 31 ====================

```
#define hardirq_trylock(cpu)    (local_irq_count(cpu) == 0)
```

与前面的 in_interrupt()比较一下就可看出，这还是在防止从一个硬中断服务程序内部调用 bh_action()。而另一道关卡 spin_trylock()就不同了，它的代码在 include/linux/spinlock.h 中：

==================== include/linux/spinlock.h 74 74 ====================

```
#define spin_trylock(lock)  (!test_and_set_bit(0,(lock)))
```

这把“锁”就是全局量 global_bh_lock，只要有一个 CPU 在 253 行至 260 行之间运行，别的 CPU 就不能进入这个区间了，所以在任何时间最多只有一个 CPU 在执行 bh 函数。这就是前述的第二条串行化措施。至于根据 bh 函数编号执行相应的函数，那就很简单了。在我们这个情景中，具体的 bh 函

数是 timer_bh()，我们将在“时钟中断”一行中阅读这个函数的代码

作为对比，我们列出另一个软中断服务程序 tasklet_action()的代码，读者可以把它与 bh_action()比较，看看有哪些重要的区别。这个函数的代码在 kernel/softirq.c 中：

==================== kernel/softirq.c 122 163 ====================

```
struct tasklet_head tasklet_vec[NR_CPUS] __cacheline_aligned;

static void tasklet_action(struct softirq_action *a)
{
    int cpu = smp_processor_id();
    struct tasklet_struct *list;

    local_irq_disable();
    list = tasklet_vec[cpu].list;
    tasklet_vec[cpu].list = NULL;
    local_irq_enable();

    while (list != NULL) {
        struct tasklet_struct *t = list;

        list = list->next;

        if (tasklet_trylock(t)) {
            if (atomic_read(&t->count) == 0) {
                clear_bit(TASKLET_STATE_SCHED, &t->state);

                t->func(t->data);
                /*
                 * talklet_trylock() uses test_and_set_bit that imply
                 * an mb when it returns zero, thus we need the explicit
                 * mb only here: while closing the critical section.
                 */
#ifdef CONFIG_SMP
                smp_mb__before_clear_bit();
#endif
                tasklet_unlock(t);
                continue;
            }
            tasklet_unlock(t);
        }
        local_irq_disable();
        t->next = tasklet_vec[cpu].list;
        tasklet_vec[cpu].list = t;
        __cpu_raise_softirq(cpu, TASKLET_SOFTIRQ);
        local_irq_enable();
    }
}
```

最后，软中断服务程序，包括 bh 函数，与常规中断服务程序的分离并不是强制性的，要根据设备

PDF-XCHANGE Click to buy NOW! www.docu-track.com

驱动的具体情况（也许还有设计人员的水平）来决定。

3.6 页面异常的进入和返回

我们在第 2 章中介绍内核对页面异常处理时，是从 do_page_fault()开始的。当时因为尚未介绍 CPU 的中断和异常机制，所以暂时跳过了对页面异常的响应过程，也就是从发生异常至 CPU 到达 do_page_fault()之间的那一段路程，以及从 do_page_fault()返回之后到 CPU 返回到用户空间这一段路程。现在，我们可以来补上这个缺口了。

与外设中断不同，各种异常都有为其保留的专用中断向量，因此相应的初始化也是直截了当的，这一点我们已经在初始化一节中看到了。

为页面异常设置的中断门指向程序入口 page_fault（见 IDT 初始化一节中所引 trap_init()中的 970 行），所以当发生页面异常时，CPU 穿过中断门以后就直接到达了 page_fault()。CPU 因异常而穿过中断门的过程，包括堆栈的变化，与因外设中断而引起的过程基本上是一样的，读者可以参阅外设中断一节。但是，有一点很重要的不同。当中断发生时，CPU 将寄存器 EFLAGS 的内容，以及代表着返回地址的 CS 和 EIP 两个寄存器的内容压入堆栈。如果 CPU 的运行级别发生变化，则在此之前还要发生堆栈的切换，并且要把代表老堆栈指针的 SS 和 ESP 的内容压入堆栈。这一点，我们已经在前面介绍过了。当异常发生时，在上述这些操作之后，还要加上附加的操作。那就是：如果所发生的异常产生出错代码的话，就把这个出错代码也压入堆栈。并非所有的异常都产生出错代码，有关详情可参考 Intel 的技术资料或相关专著，但是绝大多数异常，包括我们这里所关心的页面异常是会产生出错代码的。而且，实际上我们在第 2 章中已经看到 do_page_fault()如何通过这个出错代码识别发生异常的原因。可是，CPU 只是在进入异常时才知道是否应该把出错代码压入堆栈。而从异常处理通过 iret 指令返回时已经时过境迁，CPU 已经无从知道当初发生异常的原因，因此不会自动跳过堆栈中的这一项，而要靠相应的异常处理程序对堆栈加以调整，使得在 CPU 开始执行 iret 指令时堆栈顶部是返回地址。由于这个不同，对异常的处理和对中断的处理在代码中也要有所不同。

页面异常处理的入口 page_fault 是在 arch/i386/kernel/entry.S 中定义的：

==================== arch/i386/kernel/entry.S 410 412 ====================

```
ENTRY(page_fault)
        pushl $ SYMBOL_NAME(do_page_fault)
        jmp error_code
```

这里的跳转目标 error_code 就好像外设中断处理中的 common_interrupt 一样，是各种异常处理所共用的程序入口。而将服务程序 do_page_fault()的地址压入堆栈，则为进入具体的服务程序作好了准备。程序入口 error_code 的代码也在同一文件（arch/i386/kernel/entry.S）中：

==================== arch/i386/kernel/entry.S 295 321 ====================

```
error_code:
        pushl %ds
        pushl %eax
        xorl %eax,%eax
        pushl %ebp
```

```
300         pushl %edi
301         pushl %esi
302         pushl %edx
303         decl %eax          # eax = -1
304         pushl %ecx
305         pushl %ebx
306         cld
307         movl %es,%ecx
308         movl ORIG_EAX(%esp), %esi   # get the error code
309         movl ES(%esp), %edi         # get the function address
310         movl %eax, ORIG_EAX(%esp)
311         movl %ecx, ES(%esp)
312         movl %esp,%edx
313         pushl %esi             # push the error code
314         pushl %edx             # push the pt_regs pointer
315         movl $(__KERNEL_DS),%edx
316         movl %edx,%ds
317         movl %edx,%es
318         GET_CURRENT(%ebx)
319         call *%edi
320         addl $8,%esp
321         jmp ret_from_exception
```

读者也许注意到了，这里并不像进入中断响应时那样引用 SAVE_ALL。让我们来看看有什么区别，以及为什么。观察图 3.7，我们把 CPU 执行到这里的 307 行时的堆栈（左边）与 CPU 在外设中断时 SAVE_ALL 以后的堆栈（右边）作一比较。

顺便提一下，系统调用时的堆栈在执行完 SAVE_ALL 以后与图 3.7 的右边（中断）几乎完全一样，只是在 ORIG_EAX 位置上是系统调用号而不是中断请求号。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

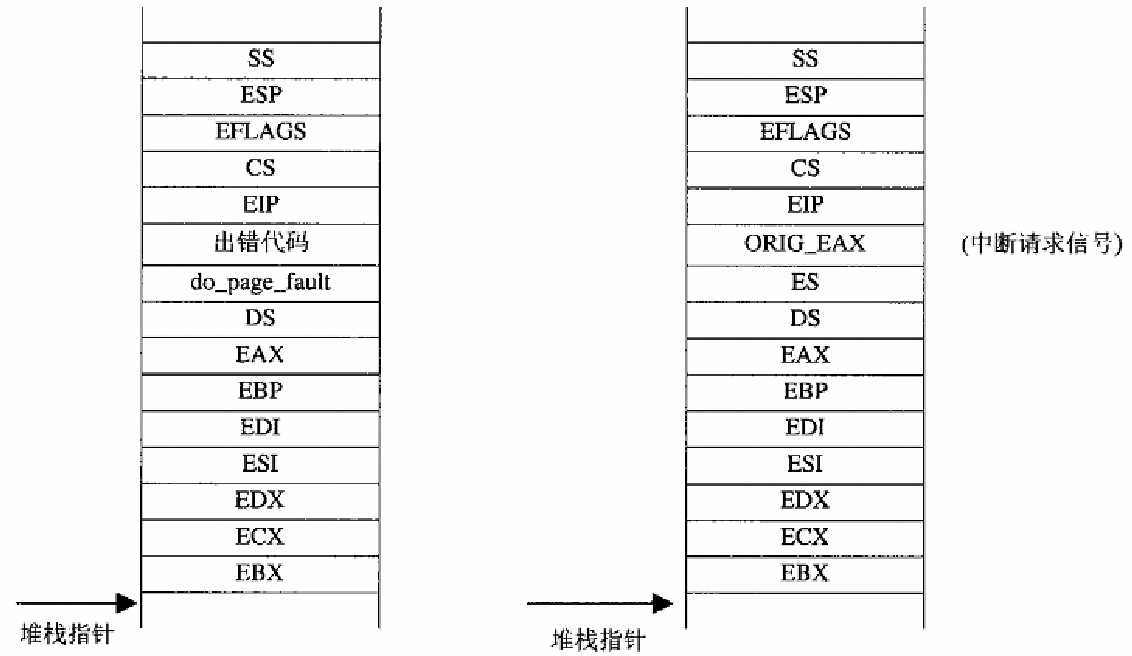


图 3.7　异常处理和中断处理系统堆栈对照图

比较之后，可以看到其实也只有在两个位置上不同。一个是与 ORIG_EAX 对应的位置上，现在是 CPU 在发生异常时压入堆栈的出错代码。另一个是在与 ES 相应的位置上，现在是 do_page_fault()的入口地址。其它就都一样了。可是，下面会将堆栈中对应于 ORIG_EAX 位置上的内容转移到寄存器%esi 中，并将其替换成%eax 中的内容。这样一来，出错代码就到了%esi 中。而堆栈中的 ORIG_EAX 就变成了-1（见 298 行和 303 行）。同时，又以寄存器%ecx 的内容替换堆栈中 ES 处的函数指针，而把函数指针转移到寄存器%edi 中。在此之前的 307 行已经将%es 的内容装入了%ecx，所以在 311 行以后函数指针 do_page_fault()在%edi 中，而堆栈中变成了寄存器%es 的副本。至此，也就是在 311 行以后，堆栈的内容与中断或系统调用时就完全一样了，只是 ORIG_EAX 的位置上为-1。这么一来，堆栈就调整好了。我们在中断一节中已经看到将来返回时在 RESTORE_ALL 中会把 ORIG_EAX 跳过去。

读者也许会问：那么，对于不产生出错代码的异常又怎么处理呢？很简单，在进入 error_code 之前补上一个就是了。请看，同一源文件（arch/i386/kernel/entry.S）。因协处理器（coprocessor）出错而导致的异常 coprocessor_error：

==================== arch/i386/kernel/entry.S 323 326 ====================

```
ENTRY(coprocessor_error)
	pushl $0
	pushl $ SYMBOL_NAME(do_coprocessor_error)
	jmp error_code
```

这里多了一行“ pushl $0”，将 0 压入堆栈中与出错代码相应的地方，此后就都一样了。

回到前面 error_code 的代码中，第 313 行和 314 行先后把%esi 和%edx 的内容压入堆栈。我们知道，%esi 中是出错代码，而 312 行已经把堆栈指针的当前内容拷贝到%edx 中。在中断一节中我们已经讲过，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

内核将 SAVE_ALL 以后堆栈中的内容视同一个 pt_regs 数据结构，而当时的堆栈指针指向该数据结构的起点。所以，这二者一项是出错代码而另一项便是 pt_regs 结构指针，这正是 do_page_fault()的两个调用参数。把调用参数压栈以后，就为 319 行的函数调用作好了准备。其它一些准备工作读者在中断响应中都已看到过，这里就不重复了。

从调用的函数，在这里是 do_page_fault()返回以后，CPU 就转入 ret_from_exception。由于 do_page_fault()的类型是 void，所以没有返回值。ret_from_exception 的代码也在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 260 279 ====================
[page_fault->error_code->...->ret_from_exception]

```
ret_from_exception:
#ifdef CONFIG_SMP
	GET_CURRENT(%ebx)
	movl processor(%ebx),%eax
	shll $CONFIG_X86_L1_CACHE_SHIFT,%eax
	movl SYMBOL_NAME(irq_stat)(,%eax),%ecx		# softirq_active
	testl SYMBOL_NAME(irq_stat)+4(,%eax),%ecx # softirq_mask
#else
	movl SYMBOL_NAME(irq_stat),%ecx	# softirq_active
	testl SYMBOL_NAME(irq_stat)+4,%ecx	# softirq_mask
#endif
	jne   handle_softirq

ENTRY(ret_from_intr)
	GET_CURRENT(%ebx)
	movl EFLAGS(%esp),%eax		# mix EFLAGS and CS
	movb CS(%esp),%al
	testl $(VM_MASK | 3),%eax	# return to VM86 mode or non-supervisor?
	jne ret_with_reschedule
	jmp restore_all
```

如果没有软中断请求需要处理，就直接进入 ret_from_intr。后面这些代码读者已经很熟悉了，要是还有困难可以回到前几节再看看。

3.7 时钟中断

在所有的外部中断中，时钟中断起着特殊的作用，其作用远非单纯的计时所能相比。当然，即使是单纯的计时也已经足够重要了。别的不说，没有正确的时间关系，你用来重建内核的工具 make 就不能正常运行了，因为 make 是靠时间标记来确定是否需要重新编译以及连接的。可是时钟中断的重要性还远不止于此。

我们在中断一节中看到，内核在每次中断（以及系统调用和异常）服务完毕返回用户空间之前都要检查是否需要调度，若有需要就进行进程调度。事实上，调度只是当 CPU 在内核中运行时才可能发生。在进程一章中，读者将会看到进程调度发生在两种情况下。一种是“自愿”的，通过像 sleep()之

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

类的系统调用实现；或者是在通过其它系统调用进入内核以后因某种原因受阻需要等待，而“自愿”让内核调度其它进程先来运行。另一种是“强制”的，当一个进程连续运行的时间超过一定限度时，内核就会强制地调度其它进程来运行。如果没有了时钟，内核就失去了与时间有关的强制调度的依据和时机，而只能依赖于各个进程的“思想觉悟”了。试想，如果有一个进程在用户空间中陷入了死循环，而在循环体内也没有作任何系统调用，并且也没有发生外设中断，那么，要是没有时钟中断，整个系统就在原地打转什么事也不能做了。这是因为，在这种情况下永远不会有调度，而死抓住 CPU 不放的进程则陷在死循环中。退一步讲，即使我们还有其它的准则（例如进程的优先级）来决定是否应该调度，那也得要有中断、异常或系统调用使 CPU 进入内核运行才能发生调度。而惟一可以预测在一定的时间内必定会发生的，就是“时钟中断”。所以，对于像 Linux 这样的“分时系统”来说，时钟中断是维护“生命”的必要条件，难怪人们称时钟中断为“heart beat”，也即“心跳”。

在初始化阶段，在对外部中断的基础设施，也就是 IRQ 队列的初始化，以及对调度机制的初始化完成以后，就轮到时钟中断的初始化。请看 init/main.c 中 start_kernel()的片段：

==================== init/main.c 534 537 ====================

```
	trap_init();
	init_IRQ();
	sched_init();
	time_init();
```

从这里也可以看出，时钟中断和调度是密切联系在一起的。以前也讲到过，一旦开始有时钟中断就可能要进行调度，所以要先完成对调度机制的初始化，作好准备。函数 time_init()的代码在 arch/i386/kernel/time.c 中：

==================== arch/i386/kernel/time.c 626 631 ====================

```
void __init time_init(void)
{
	extern int x86_udelay_tsc;

	xtime.tv_sec = get_cmos_time();
	xtime.tv_usec = 0;
```

==================== arch/i386/kernel/time.c 704 704 ====================

```
	setup_irq(0, &irq0);
  ......
```

==================== arch/i386/kernel/time.c 706 706 ====================

```
}
```

当我们提及“系统时钟”时，实际上是指着内核中的两个全局量之一。一个是数据结构 xtime，其类型为 struct timeval，是在 include/linux/time.h 中定义的：

==================== include/linux/time.h 88 91 ====================

```
struct timeval {
	time_t		tv_sec;		/* seconds */
	suseconds_t	tv_usec;	/* microseconds */
};
```

数据结构中记载的是从历史上某一刻开始的时间的“绝对值”，其数值来自计算机中一个 CMOS

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

晶片，常常称为“实时时钟”。这块 CMOS 晶片是由电池供电的，所以即使机器断了电也还能维持正确的时间。上面的 630 行就是通过 get_cmos_time()从 CMOS 时钟晶片中把当时的实际时间读入 xtime，时间的精度为秒。而时钟中断，则是由另一个晶片产生的。

另一个全局量是个无符号整数，叫 jiffies，记录着从开机以来时钟中断的次数。每个 jiffy 的长度就是时钟中断的周期，有时候也称为一个 tick，取决于系统中的一个常数 HZ，这个常数定义于 include/asm-i386/param.h 中。以后读者会看到，在内核中 jiffies 远远比 xtime 重要，是个经常要用到的变量。

系统中有很多因素会影响到时钟中断在时间上的精确度，所以要通过好多手段来加以校正。在比较新的 i386 CPU 中（主要是 Pentium 及以后），还设置了一个特殊的 64 位寄存器，称为“时间印记计数器”（Time Stamp Counter）TSC。这个计数器对驱动 CPU 的时钟脉冲进行计数，例如要是 CPU 的时钟脉冲频率为 500MHz，则 TSC 的计时精度为 2ns。由于 TSC 是个 64 位的计数器，其计数要经过连续运行上千年才会溢出。显然，可以利用 TSC 的读数来改善时钟中断的精度。不过，我们在这里并不关心时间的精度，所以跳过了代码中有关的部分，而只关注带有本质性的部分。

读者在中断一节中看到过 setup_irq()，可以回过头去看一下。这里的第一个参数为中断请求号，时钟中断的请求号为 0。第 2 个参数是指向一个 irqaction 数据结构 irq0 的指针。irq0 也是在 arch/i386/kernel/time.c 中定义的：

==================== arch/i386/kernel/time.c 547 547 ====================

```
static struct irqaction irq0  = { timer_interrupt, SA_INTERRUPT, 0, "timer", NULL, NULL};
```

可见，时钟中断的服务程序为 timer_interrupt()；中断请求 0 为时钟中断专用，因为 irq0.flags 中标志位 SA_SHIRQ 为 0；而且在执行 timer_interrupt()的过程中不容许中断，因为标志位 SA_INTERRUPT 为 1。服务程序 timer_interrupt()的代码在同一个文件（arch/i386/kernel/time.c）中：

==================== arch/i386/kernel/time.c 454 505 ====================

```
/*
 * This is the same as the above, except we _also_ save the current
 * Time Stamp Counter value at the time of the timer interrupt, so that
 * we later on can estimate the time of day more exactly.
 */
static void timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
     int count;

     /*
      * Here we are in the timer irq handler. We just have irqs locally
      * disabled but we don't know if the timer_bh is running on the other
      * CPU. We need to avoid to SMP race with it. NOTE: we don' t need
      * the irq version of write_lock because as just said we have irq
      * locally disabled. -arca
      */
     write_lock(&xtime_lock);

     if (use_tsc)
```

```
473       {
474           /*
475            * It is important that these two operations happen almost at
476            * the same time. We do the RDTSC stuff first, since it's
477            * faster. To avoid any inconsistencies, we need interrupts
478            * disabled locally.
479            */
480
481           /*
482            * Interrupts are just disabled locally since the timer irq
483            * has the SA_INTERRUPT flag set. -arca
484            */
485
486           /* read Pentium cycle counter */
487
488           rdtscl(last_tsc_low);
489
490           spin_lock(&i8253_lock);
491           outb_p(0x00, 0x43);     /* latch the count ASAP */
492
493           count = inb_p(0x40);    /* read the latched count */
494           count |= inb(0x40) << 8;
495           spin_unlock(&i8253_lock);
496
497           count = ((LATCH-1) - count) * TICK_SIZE;
498           delay_at_last_interrupt = (count + LATCH/2) / LATCH;
499       }
500
501       do_timer_interrupt(irq, NULL, regs);
502
503       write_unlock(&xtime_lock);
504
505   }
```

在这里我们并不关心多处理器 SMP 结构，也不关心时间的精度，所以实际上只剩下 501 行的 do_timer_interrupt()：

==================== arch/i386/kernel/time.c 380 386 ====================

[timer_interrupt()>do_timer_interrupt()]

```
380  /*
381   * timer_interrupt() needs to keep up the real-time clock,
382   * as well as call the "do_timer()" routine every clocktick
383   */
384  static inline void do_timer_interrupt(int irq, void *dev_id, struct pt_regs *regs)
385  {
386  #ifdef CONFIG_X86_IO_APIC
   ......
```

==================== arch/i386/kernel/time.c 400 435 ====================

```
400  #endif
```

```
401
402 #ifdef CONFIG_VISWS
403     /* Clear the interrupt */
404     co_cpu_write(CO_CPU_STAT,co_cpu_read(CO_CPU_STAT) & ~CO_STAT_TIMEINTR);
405 #endif
406     do_timer(regs);
407 /*
408  * In the SMP case we use the local APIC timer interrupt to do the
409  * profiling, except when we simulate SMP mode on a uniprocessor
410  * system, in that case we have to call the local interrupt handler.
411  */
412 #ifndef CONFIG_X86_LOCAL_APIC
413     if (!user_mode(regs))
414         x86_do_profile(regs->eip);
415 #else
416     if (!smp_found_config)
417         smp_local_timer_interrupt(regs);
418 #endif
419
420     /*
421      * If we have an externally synchronized Linux clock, then update
422      * CMOS clock accordingly every ~11 minutes. Set_rtc_mmss() has to be
423      * called as close as possible to 500 ms before the new second starts.
424      */
425     if ((time_status & STA_UNSYNC) == 0 &&
426        xtime.tv_sec > last_rtc_update + 660 &&
427        xtime.tv_usec >= 500000 - ((unsigned) tick) / 2 &&
428        xtime.tv_usec <= 500000 + ((unsigned) tick) / 2) {
429         if (set_rtc_mmss(xtime.tv_sec) == 0)
430             last_rtc_update = xtime.tv_sec;
431         else
432             last_rtc_update = xtime.tv_sec - 600; /* do it again in 60 s */
433     }
434
435 #ifdef CONFIG_MCA
   ......
```

==================== arch/i386/kernel/time.c 449 450 ====================

```
449 #endif
450 }
```

同样，我们在这里并不关心多处理器 SMP 结构中采用 APIC 时的特殊处理，也不关心 SGI 作站（402～405 行）和 PS/2 的 “ Micro channel”（435～449 行）的特殊情况，此外，我们在这里也不关心时钟的精度（420～433 行）。

这样，就只剩下了两件事。一件事是 do_timer()，另一件是 x86_do_profile()。其中 x86_do_profile() 的目的在于积累统计信息，也不是我们关心的重点。最后就只剩下 do_timer()了，那是在 kernel/timer.c 中：

==================== kernel/timer.c 674 685 ====================

[timer_interrupt()>do_timer_interrupt()>do_timer()]

```
674 void do_timer(struct pt_regs *regs)
675 {
676     (*(unsigned long *)&jiffies)++;
677 #ifndef CONFIG_SMP
678     /* SMP process accounting uses the local APIC timer */
679
680     update_process_times(user_mode(regs));
681 #endif
682     mark_bh(TIMER_BH);
683     if (TQ_ACTIVE(tq_timer))
684         mark_bh(TQUEUE_BH);
685 }
```

这里的第 676 行使 jiffies 加 1。细心的读者可能会问，为什么这里不用简单的“ jiffies++”，而要使用这么一种奇怪的方式呢？这是因为代码的作者要使将递增 jiffies 的操作在一条指令中实现，成为一个“原子”的操作。gcc 将这条语句翻译成一条对内存单元的 INC 指令。而若采用“ jiffies++”，则有可能会被编译成先将 jiffies 的内容 MOV 至寄存器 EAX，然后递增，再 MOV 回去。二者所耗费的 CPU 时钟周期几乎是相同的，但前者保证了操作的“原子”性。

函数 update_process_times()就与进程的调度有关了，我们将在进程调度一节中再来介绍。但是，从函数的名字也可以看出，它处理的是当前进程与时间有关的变量，一方面是为统计的目的，另一方面也是为调度的目的。对用于记时和统计的这些变量的操作可说是时钟中断的“前半”，可是 682 行和 684 行为时钟中断安排的“后半”和“第二职业”，却要耗费多得多的精力。

我们在前几节中已介绍过中断服务程序的“后半”，即 bh。CPU 在从中断返回之前都要检查是否在某个 bh 队列中还有事等着要处理。而这里的 682 行就通过 mark_bh()将 bh_task_vec[TIMER_BH]挂入 tasklet_hi_vec 的队列中，使 CPU 在中断返回之前执行与 TIMER_BH 对应的函数 timer_bh()，这是事先设置好了的。对此，在 kernel/sched.c 的 sched_init()中有三行重要的代码：

==================== kernel/sched.c 1260 1262 ====================

```
    init_bh(TIMER_BH, timer_bh);
    init_bh(TQUEUE_BH, tqueue_bh);
    init_bh(IMMEDIATE_BH, immediate_bh);
```

这里初始化了三个 bh。第一个显然是在每次时钟中断结束之前都要执行的，用来完成逻辑上属于时钟中断服务、但又不是那么紧急，或者可以在更为宽松的环境（开中断）下完成的操作，其相应的函数为 timer_bh()。而 TQUEUE_BH 和 IMMEDIATE_BH，则又是内核中两项重要的基础设施。我们以前讲过，Linux 内核中可能的 bh 的数量是 32。读者心里可能已经在想，32 个 bh 够吗？如果需要更多怎么办？还有，更重要地，在实践中常常会有要求让某些操作跟某个已经存在的中断服务动态地挂上钩，使一些操作按运行时的需要“挂靠”在某种中断或甚至某种其它的事件中。举例来说，如果我们要为一个外部设备写驱动程序，该设备要求每 20ms 读一次它的状态寄存器，再根据读入的信息进行某些计算，并把计算结果写入它的控制寄存器以驱动一台步进马达，而该设备并不具备产生中断的功能。其实，由于这个外设的控制完全是周期性的，本来就不必使用独立的中断，所需要解决的只是怎样与系统的时钟中断挂上钩。前面讲过，Linux 系统时钟的频率是由一个常数 HZ 决定的，定义于

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

include/asm-i386/param.h。通常 HZ 定义为 100，也即每 10ms 一次时钟中断，跟需要的 20ms 正好是整数倍关系。所以，如果写个程序，并且能在每次时钟中断中都调用它一次。而在程序中则设置一个计数器，使得每当计数为偶数时就采集数据，为奇数时就计算并输出。这样就可以解次问题了。可是，怎样让时钟中断每次都来调用它呢？TQUEUE_BH 就是为这种需要而设置的。全局量 tq_timer 指向一个队列，想要让系统在每次时钟中断时都来调用某个函数（当然是在系统空间），就将其挂入该队列里。而这里的 683 行则检查 tq_timer 是否为空。如果不为空就通过 mark_bh()把 bh_task_vec[TQUEUE_BH]也挂入 tasklet_hi_vec 的队列中，这样内核就会在执行 bh 时通过 tqueue_bh()来将该队列中所有的函数都调用一遍。由此可见，TQUEUE_BH 确实是一项很重要的基础设施。除与时钟挂钩的 tq_timer 队列外，还有其它一些 bh 和相应的队列，IMMEDIATE_BH 是其中之一。有关详情我们将在“进程”和“设备驱动”有关章节中介绍。如果说，时钟中断的“前半 ” timer_interrupt()和“后半 ” timer_bh()还是它的“正业”的话，那么 tqueue_bh()的执行便是“第二职业”了。

在做好这些准备以后，时钟中断服务的“前半”就完成了。可是读者在中断一节中已经看到，CPU 在返回途中，却在离开 do_IRQ()之前，先折入 do_softirq()去干它的“后半”和“第二职业”。在我们这个情景中，timer_bh()肯定会得到执行，而 tqueue_bh()则在 tq_timer 队列非空时会得到执行。读者也许还会问，既然 timer_bh()肯定是要执行的，为什么不干脆把它也放在 do_timer()中执行，而要费这么些周折呢？首先，前面已经看到，执行 timer_interrupt()的整个过程中中断是关闭的（见前面的 SA_INTERRUPT 标志位）；而 timer_bh()的执行则没有这么严格的要求。其次，在 do_IRQ()的代码中可以看出，对具体中断服务程序的执行与对 do_softirq()的执行不是一对一的关系。对具体中断服务程序的执行是在一个循环中进行的，而 do_softirq()只执行一次。这样，当同一中断通道内紧接着发生了好几次中断时，对 do_softirq()，从而对 timer_bh()的执行就推迟并且合并了。

与 TIMER_BH 对应的 timer_bh()在 kernel/timer.c 中：

==================== kernel/timer.c 668 672 ====================

```
void timer_bh(void)
{
	update_times();
	run_timer_list();
}
```

先看同一文件（kernel/timer.c）中的 update_times():

==================== kernel/timer.c 643 666 ====================

[timer_bh()>update_times()]

```
/*
 * This spinlock protect us from races in SMP while playing with xtime. -arca
 */
rwlock_t xtime_lock = RW_LOCK_UNLOCKED;

static inline void update_times(void)
{
	unsigned long ticks;

	/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         * update_times() is run from the raw timer_bh handler so we
         * just know that the irqs are locally enabled and so we don't
         * need to save/restore the flags of the local CPU here. -arca
         */
        write_lock_irq(&xtime_lock);

        ticks = jiffies - wall_jiffies;
        if (ticks) {
            wall_jiffies += ticks;
            update_wall_time(ticks);
        }
        write_unlock_irq(&xtime_lock);
        calc_load(ticks);
}
```

这里做了两件事。第一件事是 update_wall_time()，目的是处理所谓“实时时钟”或者说“挂钟”xtime 中的数值，包括计数，进位，以及为精度目的而作的校正。所涉及的主要也是数值的计算和处理，我们就不深入进去了。这里的 wall_jiffies 也像 jiffies 一样是个全局量，它代表着与当前 xtime 中的数值相对应的 jiffies 值，表示“挂钟”当前的读数已经较准到了时轴上的哪一点。

第二件事是 calc_load()，目的是计算和积累关于 CPU 负荷的统计信息。内核每隔 5 秒种计算、累积和更新一次系统在过去的 15 分钟、10 分钟以及 1 分钟内平均有多少个进程处于可执行状态，作为衡量系统负荷轻重的指标。由于涉及的主要是数值计算，所以我们也不深入进去了。

从 update_times()返回后，就是 timer_bh()的主体部分 run_timer_list()了。它检查系统中已经设置的各个“定时器”(timer)，如果某个定时器已经“到点”就执行为之预定的函数（这就是该定时器的 bh 函数）。我们将在“进程与进程调度”。一章中讲述定时器的设置，到那时再回过来阅读 run_timer_list()的代码。

每个定时器都由一个 timer_list 数据结构代表，定义于 include/linux/timer.h 中：

===================== include/linux/timer.h 20 25 =====================

```
struct timer_list {
        struct list_head list;
        unsigned long expires;
        unsigned long data;
        void (*function)(unsigned long);
};
```

这是一个用于链表的数据结构，链表的长度是动态的而不受限制，因此系统中可以设置的定时器数量也不受限制（早期的实现采用数组，因而受到数组大小的限制）。每个定时器都有个到点时间 expires。结构中的函数指针 function 指向预定在到点时执行的 bh 函数，并且可以带一个参数 data。（早期的实现中不能带参数）。如前所述，在执行 bh 函数时中断是打开的。

可见，在整个时钟中断服务的期间，大部分的操作是在“后半”，即 bh 函数中完成的。真正在关中断状态下执行的只是少量关键性的操作，而大量的操作尽可能要放在比较宽松的环境下，即开中断的条件下，以及允许在时间上有所伸缩的条件下完成，这样才能将对系统的影响减至最小。一方面，这应该成为系统程序设计（特别是设备驱动程序）的一项准则；而另一方面，这也对设计和开发的人

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

员提出了很高的要求，因为要区分一项操作是否必须在“前半”中执行，以及是否必须关中断，需要对系统有深刻的理解。

3.8 系统调用

如果说外部中断是使 CPU 被动地、异步地进入系统中间的一种手段，那么系统调用就是 CPU 主动地、同步地进入系统空间的手段。这里所谓“主动”，是指 CPU“自愿”的、事先计划好了的行为。而“同步”则是说，CPU（实际上是软件的设计人员）确切地知道在执行哪一条指令以后就一定会进入系统空间。相比之下，中断的发生带有很大的不可预测性。但是，尽管有着这样的区别，二者之间还是有很大的共性。这是因为，在使 CPU 的运行状态从用户态转入系统态，也就是从用户空间转入系统空间，这一个基本点上二者是一致的。当然，中断有可能发生在 CPU 已经运行在系统空间的时候，而系统调用却只发生于用户空间，这又是二者不同的地方。这里，关键是 CPU 运行状态的改变，没有了这样的手段，也就无所谓“保护模式”了。相比之下，在不分“用户态”和“系统态”的操作系统中，例如 DOS，所谓系统调用实际上只不过是动态连接的库函数调用而已。虽然在 DOS 里面系统调用也是通过中断指令 INT 来实现的，但是跟预先规定好各种函数入口地址的普通函数调用没有多大不同。如果用户程序知道具体函数的入口地址，就可以绕过“系统调用”而直接调用这些函数。

Linux 的系统调用是通过中断指令“ int $0x80”实现的。我们已经在前面几节中讨论过进程通过“陷阱门”或“中断门”进入系统空间的机制，以及 IDT 表中陷阱门的初始化。本节将着重介绍进程在系统调用中进入系统空间，以及在完成了所需的服务以后从系统空间返回的过程。这个过程并不局限于某个特定的调用，而是所有的系统调用都要经历的共同的过程。虽然我们选择了一个具体的调用作为例子，但并不从功能的角度来关心具体的调用，而是着眼于这个公共的过程。系统调用是内核所提供的最根本的、最重要的基础设施。由于系统调用与中断的共同性，读者在阅读本节时应该与前几节，特别是中断过程一节结合阅读。事实上，有些代码就是二者共用的，凡是以前已经介绍过的本节就不再重复。

由于我们并不关心内核在具体系统调用中所提供的服务，所以选择了一个非常简单的调用 sethostname()作为情景，通过对 CPU 在这个系统调用全过程中所走过的路线的分析，介绍内核的系统调用机制。

系统调用 sethostname()的功能非常简单，就是设置计算机（在网络中的）“主机名”，其使用也很简单：

```
int sethostname(cost char *name, size_t len);
```

参数 name 就是要设置的主机名，而 len 则为该字符串的长度。调用结束后返回 0 表示成功，-1 则表示失败。失败时用户程序中的全局变量 error 含有具体的出错代码。从程序设计的观点来看，Linux 的系统调用可以分成两类：一类比较接近于真正意义上的“函数”，调用的结果就是函数值，例如 getpid()就是这样；而另一类就是像 sethostname()这样的，返回的值实际上只是一个是否成功的标志，而调用的目的是通过“副作用”来体现的。但是，在 C 语言把所有可以通过调用指令来调用的程序段，也就是带有 ret 指令的程序段都称作“函数”。而中断服务程序和系统调用，由于 ret（实际上是 iret）指令

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的存在也就成了“函数”。我们在讨论中也将遵循 C 语言的规定和传统一概称之为函数。

为了帮助读者更好地理解系统调用的全过程，我们从用户空间对函数 sethostname()的调用开始我们的情景分析。其实，sethostname()是一个库函数（在/usr/lib/libc.a），而实际的系统调用就是在那个函数中发出的。GNU 的 C 语言库函数的源代码也是公开的，可以从 GNU 的网站下载。但是，我们在这里采用从 libc.a 反汇编得到的代码。原因是，一来方便，“得来全不费工夫”；二来，读者多接触一些汇编代码也是有好处的。特别是对于系统程序员本说，阅读和使用汇编语言也是一种有用的技能。

```
sethostname.o:     file format elf32-i386

Disassembly of section .text:

00000000 <sethostname>:
   0:89 da                mov    %ebx,%edx
   2:8b 4c 24 08              mov    0x8(%esp,1),%ecx
   6:8b 5c 24 04              mov    0x4(%esp,1),%ebx
   a:b8 4a 00 00 00           mov    $0x4a,%eax
   f:cd 80                int    $0x80
  11:     89 d3               mov    %edx,%ebx
  13:     3d 01 f0 ff ff          cmp    $0xfffff001,%eax
  18:     0f 83 fc ff ff ff       jae    1a <sethostname+0x1a>
  1e:     c3                  ret
```

进入函数 sethostname()以后，堆栈指针%esp 指向返回地址，而在堆栈指针的内容加 4 的地方则是调用该函数时的第一个参数（name），加 8 的地方为第二个参数 len，依次类推。由于 i386 运行于 32 位模式，所有的参数都是按 32 位长整数压入堆栈的。指令“ movl 0x8(%esp, 1), %ecx”表示将相对于寄存器%esp 的位移为 0x8（位移单位为 1）处的内容（在我们这个情景中就是参数 len）存入寄存器%ecx。然后，又将参数 name 从堆栈中存入寄存器%ebx。最后是将代表 sethostname()的系统调用号 0x4a 存入寄存器%eax，接着就是中断指令“ int $0x80”。这里，读者已经看到，Linux 内核在系统调用时是通过寄存器而不是通过堆栈传递参数的。

为什么要用寄存器传递参数？读者也许还记得：当 CPU 穿过陷阱门，从用户空间进入系统空间时，由于运行级别的变动，要从用户堆栈转换到系统堆栈。如果在 INT 指令之前把参数压入堆栈，那是在用户堆栈中，而进入系统空间以后就换成了系统堆栈。虽然进入系统空间之后也还可以从用户堆栈中读取这些参数，但毕竟比较费事了。而通过寄存器来传递参数，则读者下面会看到，是个巧妙的安排。我们暂且不随着 CPU 进入内核，而先看一下从系统调用返回以后的情况。首先是从%edx 中恢复%ebx 原先的内容，那是在系统调用之前保存在%edx 中的（%edx 中原先的内容就丢失了，这是一种约定，gcc 在使用寄存器时会遵守这个约定）。然后就是检查系统调用的返回值，那是在寄存器%eax 中。如果%eax 中的内容是在 0xfffff001 与 0xffffffff 之间，也就是-1 至-4095 之间，那就是出错了，就要转向__syscall_error()并从那里返回。这里的 1a:R_386_PC32 表示地址 sethostname+0x1a 处为重定位信息，在连接时会把地址__syscall_error()填入该处。函数__syscall_error()也在 libc.a 中：

```
sysdep.o:     file format elf32-i386

Disassembly of section .text:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

00000000 <__syscall_error>:
   0:f7 d8               neg    %eax

00000002 <__syscall_error_1>:
   2:50                  push   %eax
   3:e8 fc ff ff ff           call   4 <__syscall_error_1+0x2>
   8:59                  pop    %ecx
   9:89 08               mov    %ecx,(%eax)
   b:b8 ff ff ff ff           mov    $0xffffffff,%eax
  10:     c3                  ret


errno-loc.o:     file format elf32-i386

Disassembly of section .text:

00000000 <__errno_location>:
   0:55                  push   %ebp
   1:89 e5               mov    %esp,%ebp
   3:83 ec 08                 sub    $0x8,%esp
   6:a1 00 00 00 00           mov    0x0,%eax
   b:85 c0               test   %eax,%eax
   d:75 07               jne    16 <__errno_location+0x16>
   f:b8 00 00 00 00           mov    $0x0,%eax
  14:     c9                  leave
  15:     c3                  ret
  16:     e8 fc ff ff ff           call   17 <__errno_location+0x17>
  1b:     8b 80 b8 01 00 00        mov    0x1b8(%eax),%eax
  21:     eb f1               jmp    14 <__errno_location+0x14>
```

在__syscall_error 中，先取%eax 内容的负值，使其数值变成 1～4095 之间，这就是出错代码，并将其压入堆栈。接着，又调用__errno_location()，将全局量 errono 的地址取入%eax。然后从堆栈中抛出出错代码至%ecx、并将其写入全局量 errono。最后，在返回之前，将%eax 的内容改成-1。这样，通过寄存器%eax 返回到用户进程的数值便是-1，而 errono 则含有具体的出错代码。这是对大部分系统调用（返回整数的调用）返回值的约定。

搞清了发生在用户空间的过程，我们就进入内核，也就是系统空间中去了。CPU 穿过陷阱门的过程与发生中断时穿过中断门的过程相同，这里就不重复了。不过，还是要指出，因外部中断而穿过中断门时是不检查中断门所规定的准入级别的，而在通过 INT 指令穿越中断门或陷阱门时，则要核对所规定的准入级别与 CPU 的当前运行级别。为系统调用设置的陷阱门的准入级别 DPL 为 3。寄存器 IDTR 指向当前的中断向量表 IDT，而 IDT 中对应于 0x80 的表项就是为 INT $0x80 设置的陷阱门，其中的函数指针指向 system_call()。当 CPU 到达 system_call()时，已经从用户态切换到了系统态，并且从用户堆栈换成了系统堆栈，相当于 CPU 在发生于用户空间的外部中断过程中到达 IRQ0xYY_interrupt 时的状态，读者不妨先回过头去重温一下。

如前所述，CPU 在穿过陷阱门进入系统内核时并不自动关中断，所以系统调用的过程是可中断的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

函数 system_call()的代码在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 195 205 ====================

```
195  ENTRY(system_call)
196         pushl %eax                # save orig_eax
197         SAVE_ALL
198         GET_CURRENT(%ebx)
199         cmpl $(NR_syscalls),%eax
200         jae badsys
201         testb $0x02,tsk_ptrace(%ebx)     # PT_TRACESYS
202         jne tracesys
203         call *SYMBOL_NAME(sys_call_table)(,%eax,4)
204         movl %eax,EAX(%esp)          # save the return value
205  ENTRY(ret_from_sys_call)
```

首先是将寄存器%eax 的内容压入堆栈。系统堆栈中的这个位置在代码中称为 orig_ax，在外部中断过程中用来保存（经过变形的）中断请求号，而在系统调用中则用来保存系统调用号。SAVE_ALL 我们已经在中断过程一节中看到过了。但是，这里要指出，对于压入堆栈中的寄存器内容的使用方式是不一样的。在中断过程中，SAVE_ALL 以后，当调用具体的中断服务程序时已经保存在堆栈中的内容是作为一个 pt_regs 数据结构，当成参数传递给 do_IRQ()，然后又传递给具体的服务程序的，这一点读者在中断服务一节中已经看到。可是，在系统调用中就不同了，这里堆栈中每个寄存器的内容可以根据需要作为独立的参数传递给具体的服务程序。以 sethostname()为例，需要传递的参数是两个，分别在%ebx 和%ecx 中。在 SAVE_ALL 中%ebx 是最后压入堆栈的，%ecx 次之。所以堆栈中%ebx 的内容就成为参数 1，而%ecx 的内容就是参数 2 了。回到 SAVE_ALL 去看一下，可以看到被压入堆栈的寄存器依次为。%es、%ds、%eax、%ebp。%edi、%esi、%edx、%ecx 和%ebx。这里的%eax 持有系统调用号（与 orig_ax 相同），显然不能再用来传递参数；而%ebp 是用作子程序调用过程中的“帧”（frame）指针的，也不能用来传递参数。这样，实际上就只有最后 5 个寄存器可以用来传递参数，所以，在系统调用中独立传递的参数不能超过 5 个。从这里也可以看出，SAVE_ALL 中将寄存器压入堆栈的次序并不是随意决定的，而有其特殊的考虑。

宏调用 GET_CURRENT(%ebx)使寄存器%ebx 指向当前进程的 task_struct 结构（关于 GET_CURRENT 我们将在进程一章中介绍）。然后，就检查寄存器%eax 中的系统调用号是否超出了范围。在 task_struct 数据结构中有个成分 flags，其中有个标志位叫 PT_TRACESYS。一个进程可以通过系统调用 ptrace()，将一个子进程的 PT_TRACESYS 标志位设成 1，从而跟踪该子进程的系统调用。Linux 系统中有一条命令 strace 就是干这件事的，是一个很有用的工具。这里 system_call()中的第 201 行就是在检查当前进程的 PT_TRACESYS 是否为 1。注意，flags(%ebx)并不是一个函数调用，而是表示相对于%ebx 的内容，也就是当前进程的 task_struct 结构指针、位移为 flags 处的地址，而 flags 在 arch/i386/kernel/entry.S 中的 75 行定义为 4。这一点以前已经讲过，这里再提醒一下。

当 PT_TRACESYS 标志位（0x20）为 1 时，就要转入 tracesys，其代码也在 arch/i386/kernel/entry.S 中：

==================== arch/i386/kernel/entry.S 244 254 ====================

```
tracesys:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	movl $-ENOSYS,EAX(%esp)
	call SYMBOL_NAME(syscall_trace)
	movl ORIG_EAX(%esp),%eax
	cmpl $(NR_syscalls),%eax
	jae tracesys_exit
	call *SYMBOL_NAME(sys_call_table)(,%eax,4)
	movl %eax,EAX(%esp)		# save the return value
tracesys_exit:
	call SYMBOL_NAME(syscall_trace)
	jmp ret_from_sys_call
```

将这一段程序与前面正常执行时的 203 行作一比较，就可以看到不同之处在于：当 PT_TRACESYS 为 1 时，在调用具体的服务程序之前和之后都要调用一下函数 syscall_trace()，向父进程报告具体系统调用的进入和返回。我们将在讲述进程间通信时再深入到 syscall_trace()中去，但是有兴趣的读者不妨先自己看看。现在回到 system_call()中继续看那里的 203 行。这是一条 call 指令，所 call 的地址在一个函数指针中，而这个函数指针在数组 sys_call_table[]中以%eax 的内容为下标、单位为 4 个字节的元素中。表达式(,%eax,4)的第一个逗号前面为空，表示在%eax 的基础上并没有其它的位移，而 4 则表示计算位移（%eax 相对于 sys_call_table）时的单位为 4 字节。系统调用跳转表 sys_call_table[]是一个函数指针数组，由于篇幅较大，我们把它单独作为一节，附在本章之后。

表中凡是内核不支持的系统调用号全部都指向 sys_ni_syscall()，这个函数只是返回一个出错代码：-ENOSYS，表示该系统调用尚未实现。结合前面讲过的 libc.a 中的处理，可知此时用户程序会得到返回值-1，而全局量 errno 的值为 ENOSYS

跳转表中位移为 0x4a，也就是 74 处的函数指针（见后面跳转表中的 500 行）为 sys_sethostname，所以在我们这个情景中就进入了 sys_sethostname()，这也是在 kernel/sys.c 中定义的：

==================== kernel/sys.c 971 987 ====================

```
asmlinkage long sys_sethostname(char *name, int len)
{
	int errno;

	if (!capable(CAP_SYS_ADMIN))
		return -EPERM;
	if (len < 0 || len > __NEW_UTS_LEN)
		return -EINVAL;
	down_write(&uts_sem);
	errno = -EFAULT;
	if (!copy_from_user(system_utsname.nodename, name, len)) {
		system_utsname.nodename[len] = 0;
		errno = 0;
	}
	up_write(&uts_sem);
	return errno;
}
```

可想而知，sethostname 应该是只有特权用户才可以进行的操作，所以一上来就先检查这一点。函数 capable(CAP_SYS_ADMIN)检查当前进程是否享有 CAP_SYS_ADMIN 的授权。如没有的话就返回

出错代码-EPERM。然后，又对字符串的长度进行检查以保证安全。

在多处理器系统中，同时可以有多个进程在不同的 CPU 上运行。这样，就有可能发生两个进程同时调用 sethostname()，而形成这样的现象：

(1) 进程 A 调用 sethostname()，要把主机名设成“ AB”。
(2) 进程 C 在另一个 CPU 上运行，也调用 sethostname()，要把主机名设成“ CD”。
(3) 进程 A 先进入内核，并且已经在 sys_setthostname() 中将“ A ”写入了内核中的 system_utsname.nodename，可是还没有来得及写“ B”之前发生了中断，而 C 在这个时候插了进来。
(4) 进程 C 进入内核，并且完成了对 sethostname() 的调用，成功地将内核中的 system_utsname_nodename 设置成“ CD”。
(5) 稍后，进程 A 恢复运行，继续把“ B”写入 system_utsname.nodename。
(6) 当进程 A 完成对 sethostname()的调用而“成功”返回时，内核 system_utsname.nodename 的内容却是“ CB”。

在操作系统理论中，这种现象称为“ race condition”（抢道）。为了防止这种情况发生，就要将对 system_utsname.nodenamer 的操作放在受到“信号量”（semaphore）保护的“临界区”中，而 sys_sethostname()中 979 行的 down_write()和 985 行的 up_write()所实现的正是这样的保护机制。有了这种保护，上述过程中当进程 C 到达 979 行时会发现已经有个进程正在里面操作，“请勿打扰”，而自愿暂缓，让别的进程先运行，从而避免了互相抢道。

下面，就是本次系统调用所要完成的实质性的操作了，这就是将参数 name 所指向的字符串写入内核中的 system_utsname.nodename。这个操作的源在用户空间中，而目标在系统中间中，所以要通过一个宏操作 copy_from_user()来完成复制。如前所述，系统调用时是通过寄存器传递参数的，能够通过寄存器传递的信息量显然不大，所以传递的参数大多是指针，这样才能通过指针找到更大块的数据。因此，对于系统调用的实现，类似于 copy_from_user()这样在用户空间和系统空间之间复制数据的操作是很重要、也很常用的。对于 i386 CPU，宏操作 copy_from_user()是在 include/asm-i386/uaccess.h 中定义的：

==================== include/asm-i386/uaccess.h 567 570 ====================

```
#define copy_from_user(to,from,n)              \
    (__builtin_constant_p(n) ?          \
     __constant_copy_from_user((to),(from),(n)) :  \
     __generic_copy_from_user((to),(from),(n)))
```

当复制的长度为一些特殊的常数，例如 4、8、…、512 等等时，具体的操作要略为简单一些，而在一般的情况下则通过__generic_copy_user()来完成。其代码在 arch/i386/lib/usercopy.c 中：

==================== arch/i386/lib/usercopy.c 50 56 ====================

```
unsigned long
__generic_copy_from_user(void *to, const void *from, unsigned long n)
{
    if (access_ok(VERIFY_READ, from, n))
        __copy_user_zeroing(to,from,n);
    return n;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

对于读操作，access_ok()只是检查参数 from 和 n 的合理性，例如(from + n)是否超出了用户空间的上限，而并不检查该区间是否已经映射。然后，就通过另一个宏操作__copy_user_zeroing()从用户空间复制。这里__copy_user_zeroing()的代码可以说是一块“硬骨头”。可是，这个操作对于系统调用又是很重要的。而且还有一些其它的类似操作，例如在 copy_to_user()中调用的__copy_user()以及__constant_copy_user()，还有__do_strncpy_form_user()，get_user()等等都与此非常相似，所以还是值得“啃”一下的。另一方面，我们在第 2 章中讲述 do_page_fault()留下了一个尾巴，正是跟这些操作有关的。宏操作__copy_user_zeroing()的定义在 include/asm-i386/uaccess.h 中：

==================== include/asm-i386/uaccess.h 263 289 ====================

```
263  #define __copy_user_zeroing(to,from,size)                \
264  do {                                                     \
265       int __d0, __d1;                                     \
266       __asm__ __volatile__(                               \
267            "0:  rep; movsl\n"                             \
268            "    movl %3,%0\n"                             \
269            "1:  rep; movsb\n"                             \
270            "2:\n"                                         \
271            ".section .fixup,\"ax\"\n"                     \
272            "3:  lea 0(%3,%0,4),%0\n"                      \
273            "4:  pushl %0\n"                               \
274            "    pushl %%eax\n"                            \
275            "    xorl %%eax,%%eax\n"                       \
276            "    rep; stosb\n"                             \
277            "    popl %%eax\n"                             \
278            "    popl %0\n"                                \
279            "    jmp 2b\n"                                 \
280            ".previous\n"                                  \
281            ".section __ex_table,\"a\"\n"                  \
282            "    .align 4\n"                               \
283            "    .long 0b,3b\n"                            \
284            "    .long 1b,4b\n"                            \
285            ".previous"                                    \
286            : "=&c"(size), "=&D" (__d0), "=&S" (__d1)      \
287            : "r"(size & 3), "0"(size / 4), "1"(to), "2"(from) \
288            : "memory");                                   \
289  } while (0)
```

首先来看__copy_user_zeroing()代码中常规的部分，这些代码是在操作顺利，一切都正常的情况下执行的。这一部分实质上只有 267～270 四行，加上 286～288 三行。286 行为“输出部”，共说明了三个变量，分别为%0、%1 以及%2。其中%0 对应于参数 size，与寄存器%%ecx 结合；%1 对应于局部变量__d0，与寄存器%%edi 结合；而%2 则对应于局部变量__d1，与寄存器%%esi 结合。287 行为“输入部”，说明了四个变量。第一个为%3，是一个寄存器变量，初值为(size&3)，而后面两个则分别等价于%1，%2 和%3，分别应该置初值为(size/4)，参数 to，以及参数 from。完成了输入部所规定的初始化以后，就开始执行 267～270 行的汇编语言程序。程序中利用了 X86 处理器的 REP 和 MOVS 指令进行成

PDF-XCHANGE Click to buy NOW! www.docu-track.com

串 MOVE；寄存器%%ecx 为计数器，%%esi 为源指针，%%edi 为目标指针。先按长整数进行，然后对剩余的部分（不超过 3 个）字节按字节进行。如果用 C 语言来写这段程序，那就相当于：

```
__copy_user_zeroing(void *to, void *from, size)
{
     int r;
     r = size & 3
     size = size/4;
     while(size--) *((int *) to)++ = *((int *) from)++;
     while(r--) *((char *) to)++ = *((char *) from)++;
}
```

显然，二者的效率是不能相比的。读者在前几节中已经看到过类似的代码，所以这一部分代码是容易理解的。

可是，为什么要有从 271 行至 280 行这些代码呢？代码的作者特地写了个说明，就是文件“Documentation/exception.txt”，解释其原因（如果读者的计算机安装了 Linux，可以在/usr/src/linux/Documentation 目录中找到这个文件）。不过读者在阅读那篇说明时可能还会感到困难，所以我们结合本节的情景分析加以补充说明。当内核从一个进程得到从用户空间传递进来的指针时，就像这个情景中的 name，是很难保证这个指引的“合法”性的，更难保证在长度为 len 的整个区间都是“合法”的。所以，为安全起见应该先检查这个区间的合法性，看看由指针和长度两个参数所决定的虚存区间是否已经建立映射。每个进程都有个代表它的虚存空间的 mm_struct 数据结构，记录着该进程在用户空间所有已经建立映射的区间。只要搜索这个数据结构中的链表，就可以发现从 name 开始，长度为 len 的区间是否已经建立映射，并且是否允许所需的操作（读或写）。内核中专门有个函数 verify_area()用于这个目的。而 Linux 内核老一些的版本中确实就是这样做的。但是，每次从用户区读或写时都要进行这样的检查实在是个负担，测试表明这个负担在典型的应用中确实显著地影响了效率。在实际应用中，虽然指针有问题的可能性也是有的，甚至可能还不小，但毕竟总是少数，也许可以说“百分之九十五以上的指引都是好的”，实在犯不着为少数的坏指针而“打击一大片”，致使总体效率下降。所以，新版本就决定把对指针合法性的检查取消了。万一碰上了坏指针，那就让页面异常发生吧，内核可以在页面异常的服务程序中个别地处理这个问题。

现在，我们再回过头去看看 do_page_fault()。当碰上坏指针而页面异常真的发生时，在 do_page_fault()中，首先就是通过 find_vma()搜索当前进程的虚存区间链表，如果搜索失败就转入 bad_area。在第 2 章中，我们对于 bad_area 只讲了当异常发生于 CPU 运行在用户空间时的情况。而在我们现在这个情景中，则异常发生于当 CPU 运行在系统空间的时候。虽然访问失败的目标地址在用户空间中，但 CPU 的“执行地址”却是在系统空间中。为方便起见，我们再列出 do_page_fault()中有关的几行代码：

==================== arch/i386/mm/fault.c 299 299 ====================

[do_page_fault()]

```
299  do_sigbus:
   ......
```

==================== arch/i386/mm/fault.c 315 318 ====================

```
315       /* Kernel mode? Handle exceptions or die */
316       if (!(error_code & 4))
317            goto no_context;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        return;
```

==================== arch/i386/mm/fault.c 255 260 ====================

```
no_context:
        /* Are we prepared to handle this kernel fault?  */
        if ((fixup = search_exception_table(regs->eip)) != 0) {
                regs->eip = fixup;
                return;
        }
```

就是说，如果内核能够在一个“异常表”中找到发生异常的指令所在的地址，并得到相应的“修复”地址 fixup，就将 CPU 在异常返回后将要重新执行的地址替换成这个“修复”地址。为什么要这样做呢？因为在这种情况下内核不能为当前进程补上一个页面（那样的话 name 所指的字符串就变成空白了）。而如果任其自然的话，则从异常返回以后，当前进程必然会接连不断地因执行同一条指令而产生新的异常，落入“万劫不复”的地步。所以，必须把它“从泥坑里拉出来”。函数 search_exception_table() 是在 arch/i386/mm/extable.c 中定义的：

==================== arch/i386/mm/extable.c 33 55 ====================

[do_page_fault()>search_exception_table()]

```
33   unsigned long
34   search_exception_table(unsigned long addr)
35   {
36           unsigned long ret;
37
38   #ifndef CONFIG_MODULES
39           /* There is only the kernel to search.  */
40           ret = search_one_table(__start___ex_table, __stop___ex_table-1, addr);
41           if (ret) return ret;
42   #else
43           /* The kernel is the last "module" -- no need to treat it special.  */
44           struct module *mp;
45           for (mp = module_list; mp != NULL; mp = mp->next) {
46                   if (mp->ex_table_start == NULL)
47                           continue;
48                   ret = search_one_table(mp->ex_table_start,
49                                          mp->ex_table_end - 1, addr);
50                   if (ret) return ret;
51           }
52   #endif
53
54           return 0;
55   }
```

不管 38 行的 CONFIG_MODULES 是否有定义，即是否支持“可安装模块”（取决了系统配置），最终总是要调用 search_one_table()。那也是在同一个源文件 arch/i386/mm/extable.c 中：

==================== arch/i386/mm/extable.c 12 31 ====================

[do_page_fault()>search_exception_table()>search_one_table()]

```
static inline unsigned long
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
search_one_table(const struct exception_table_entry *first,
           const struct exception_table_entry *last,
           unsigned long value)
{
        while (first <= last) {
           const struct exception_table_entry *mid;
           long diff;

           mid = (last - first) / 2 + first;
           diff = mid->insn - value;
                if (diff == 0)
                        return mid->fixup;
                else if (diff < 0)
                        first = mid+1;
                else
                        last = mid-1;
        }
        return 0;
}
```

显然，这里所实现的是在一个 exception_table_entry 结构数组中进行的对分搜索。数据结构 struct exception_table_entry 又是在 include/asm-i386/uaccess.h 中定义的：

==================== include/asm-i386/uaccess.h 67 83 ====================

```
/*
 * The exception table consists of pairs of addresses: the first is the
 * address of an instruction that is allowed to fault, and the second is
 * the address at which the program should continue.  No registers are
 * modified, so it is entirely up to the continuation code to figure out
 * what to do.
 *
 * All the routines below use bits of fixup code that are out of line
 * with the main instruction path.  This means when everything is well,
 * we don't even have to jump over them.  Further, they do not intrude
 * on our cache or tlb entries.
 */

struct exception_table_entry
{
     unsigned long insn, fixup;
};
```

结构中的 insn 表示可能产生异常的指令所在的地址，而 fixup 则为用来替换的“修复”地址。读者会问：可能发生问题的指令有那么多，怎么能为每一条可能发生问题的指令都建立这样一个数据结构呢？回答是：首先，可能发生问题的指令其实并不像想像的那么多；其次，由谁来为这此指令建立这样的数据结构呢？很简单，就是“谁使用，谁负责”。例如，我们这里的__copy_user_zeroing()要从用户空间拷贝，可能发生问题，它就应该负责在异常表中为其可能发生问题的指令建立起这样的数据结构。

现在我们可以回到__copy_user_zeroing()的代码中了。首先，在这里可能发生问题的指令其实只有

PDF-XCHANGE Click to buy NOW! www.docu-track.com

两条，一条是 267 行标号为 0 的 movsl，另一条则是 269 行标一号为 1 的 movsb。所以应该建立两个表项，这就是 282 行至 284 行所说明的，关键之处在 283 行和 284 行。283 行表示，如果异常发生在前面标号为 0 处的地址，也就是指令 movsl 所在的地址，那么其“修复地址 ” fixup 为前面标号为 3 处的地址，也就是指令 lea 所在的地址。这时，CPU 从“修复地址”开始做些什么修复呢？在这里是通过 stosb 把 system_utsname.nodename 中剩余的部分设成 0（当然也可以是什么都不做）。然后，就通过 279 行的 JMP 指令跳转到前面标号为 2 处，也就是结束的地方。这样，虽然从用户空间拷贝的目的没有达到，却避免了陷入在“异常—重执行”之间可能发生的无限循环。

大家知道，程序经编译（或汇编）连接以后，其可执行代码分成 text 和 data 两个段。但是，其实 GNU 的 gcc 和 ld 还支持另外两个段。一个是 fixup，专门用于异常发生后的修复，实际上跟 text 段没有太大区别。另一个是__ex_table，专门用于异常地址表。而__copy_user_zeroing()中的 271 行和 281 行就是告诉 gcc 应该把相应的代码分别放在 fixup 和__ex_table 段中，连接时 ld 会按地址排序将这些表项装入异常地址表中。

实际上，不光是像__copy_user_zeroing()这样的函数要准备好“修复地址”，任何在内核中运行时可能发生问题的都要有所准备，其中还包括我们在前一节中看到过的 RESTORE_ALL。当时为了让读者把注意力集中在中断的基本机制上而没有讲述有关的内容，我们在下面讲到从系统调用返回时会加以补充。这里，读者还应注意一下函数__generic_copy_from_user()的返回值。从代码中可以看到，返回的是调用参数，也就是从用户空间拷贝的长度。这是怎么回事呢？这是因为__copy_user_zeroing()不是一个函数，而是一个宏定义。在执行的过程中，n 随着复制而减小，一直到 0 为止。如果中途失败的话，则 n 代表了剩下未完成部分的大小。回头看一下__copy_user_zeroing()中的第 273 行，这里的%0 就是参数 size；因而也就是 n。同时，它就是寄存器%%ecx。在 movsl 或 movsb 执行的过程中，%%ecx 的值一直减小，直到为 0 时 movsl 或 movsb 就结束了。当操作中途失败而到达 273 行时，%%ecx 的值一定是非 0。可是，下面在 276 行还要用%%ecx，所以先把它保存在堆栈中，而到 278 行再来恢复。所以，最后在__generic_copy_from_user()中返回的 n 表示还有几个字节尚未完成。而在 sys_sethostname()中，则根据这个返回值来判断 copy_from_user()是否成功。当返回值为 0 时，就把 errno 也设成 0。这样最后 sys_sethostname()返回 0 表示成功，而若在 copy_from_user()过程中失败则返回-EFAULT。

由于 sys_sethostname()本身很简单，现在回到本节开头的 system_call()。CPU 从具体系统调用的服务程序返回时，由服务程序准备好的返回值在寄存器%eax 中，所以在第 204 行将它写入到堆栈中与%eax 对应的地方，这样在 RESTORE_ALL 以后，这个返回值仍通过%eax 传回用户空间。这以后，CPU 就到达了 ret_from_sys_call。

==================== arch/i386/kernel/entry.S 205 223 ====================

kystel：11＊call－＞rerfrofrotlT＊syscalu

```
ENTRY(ret_from_sys_call)
#ifdef CONFIG_SMP
     movl processor(%ebx),%eax
     shll $CONFIG_X86_L1_CACHE_SHIFT,%eax
     movl SYMBOL_NAME(irq_stat)(,%eax),%ecx          # softirq_active
     testl SYMBOL_NAME(irq_stat)+4(,%eax),%ecx # softirq_mask
#else
     movl SYMBOL_NAME(irq_stat),%ecx       # softirq_active
     testl SYMBOL_NAME(irq_stat)+4,%ecx    # softirq_mask
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#endif
        jne   handle_softirq

ret_with_reschedule:
        cmpl $0,need_resched(%ebx)
        jne reschedule
        cmpl $0,sigpending(%ebx)
        jne signal_return
restore_all:
        RESTORE_ALL
   ......
```

==================== arch/i386/kernel/entry.S 282 284 ====================

```
handle_softirq:
        call SYMBOL_NAME(do_softirq)
        jmp ret_from_intr
```

读者已经读过从中断返回时的代码，对上面这些代码应该不会有问题了。

需要补充的是，在 RESTORE_ALL 中有三条指令可能会引起异常，所以需要为之准备“修复”。这三条指令是：popl %ds，popl %es 以及 iret。我们先看代码（arch/i386/kernel/entry.S），再加以讨论：

==================== arch/i386/kernel/entry.S 101 130 ====================

```
#define RESTORE_ALL     \
        popl %ebx;     \
        popl %ecx;     \
        popl %edx;     \
        popl %esi;     \
        popl %edi;     \
        popl %ebp;     \
        popl %eax;     \
1:      popl %ds;\
2:      popl %es;\
        addl $4,%esp; \
3:      iret;          \
.section .fixup,"ax";   \
4:      movl $0,(%esp);    \
        jmp 1b;        \
5:      movl $0,(%esp);    \
        jmp 2b;        \
6:      pushl %ss;     \
        popl %ds;\
        pushl %ss;     \
        popl %es;\
        pushl $11;     \
        call do_exit; \
.previous;             \
.section __ex_table,"a";\
        .align 4;\
        .long 1b,4b;  \
        .long 2b,5b;  \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
129         .long 3b,6b;   \
130  .previous
```

这里准备了三个“修复”地址，分别在 127～129 行；而可能出问题的指令则分别在 109 行、110 行和 112 行。那么，为什么从堆栈中恢复%ds 会有可能发生问题呢？读者也许还记得，每当装入一个段寄存器时，CPU 都要根据这新的段选择码以及 GDTR 或 LDTR 的内容在相应的段描述表中找到所选择的段描述项，并加以检查。如果描述项与选择码都有效并且相符，就将描述项装入到 CPU 中段寄存器的“不可见”部分，使得以后不必每次都要到内存中去访问该描述项。可是，如果因为不管什么原因而使得选择码或描述项无效或不符时，CPU 就会产生一次“全面保护”（General Protection）异常（称为 GP 异常）。当这样的异常发生于系统空间时，就要为之准备好修复手段。在这里，为“ popl %ds ”准备的修复手段是从标号为 4 处，即 114 行的“ move $0, (%esp)”指令开始的程序段，实际上只有两行。这条指令将%ds 在堆栈中的副本先清成 0，然后在 115 行转回 109 行重新执行“ popl %ds”。为什么这样就能“修复”呢？其实并不是真的修复，而只是避免进一步的 GP 异常。以 0 作为段选择码称为“空选择码”。将空选择码装入一个段寄存器（除 CS 和 SS 以外）本身不会引起 GP 异常，而要到以后企图通过这个空选择码访问内存时才会引起异常，但那是回到用户空间以后的事了。在用户空间发生异常，最多也不过是把这进程“杀”了，而不会在系统一级上产生问题。所以，这里的修复手段实际上是把问题往下推、往后推而已。110 行的“ popl %es”与此相同。

最后；为什么“ iret”也可能发生问题，又怎样“修复”呢？当 i386 CPU 从系统空间中断返回到用户空间时，要从系统堆栈中恢复用户堆栈的指针，包括堆栈段寄存器的内容，并从系统堆栈中恢复在用户空间的返回地址，包括代码段寄存器的内容。与数据段寄存器%ds 类似，这两个步骤都有可能发生问题而产生 GP 异常，使 CPU 回不到用户中间中去。那么，怎样修复呢？对 CS 和 SS 不能通过使用空选择码的“瞒天过海”手段，因为 CS 和 SS 根本不接受空选择码（会产生 GP 异常）。所以，问题比“ popl %ds”所可能发生的问题更为严重。而解决的办法，则只好通过 do_exit()（详见“进程与进程调度”一章），将当前进程“丢卒保车”杀掉算了（见 118～123 行）。把当前进程杀了以后，内核会调度另一个进程成为当前进程。所以，当再要从系统空间返回到用户空间时，是返回到另一个进程的用户空间中去，那时候要从系统堆栈中恢复的寄存器副本也是另一个进程的副本了。

系统调用 sethostname()的实现虽然很简单，但是从内核中的入口 system_call 到进入 sys_sethostname()前的这一段代码，以及从 sys_sethostname()返回后直到完成 RESTORE_ALL 中的 iret 指令这一段代码，则是所有系统调用所共用的。不管什么系统调用，其进入内核以及退出内核的过程都是相同的。以后，当我们谈到系统调用时，就直接从内核中的实现，如 sys_sethostname()那样开始。

最后，还要指出一个读者已经看到但是未必清楚地意识到的事实，那就是从内核中可以直接访问当前进程的用户空间，所使用的虚拟地址也与当进程处于用户空间时的地址完全相同。当然，反过来就不可以了。

3.9　系统调用号与跳转表

文件 include/asm-i386/unistd.h 上为每个系统调用定义了一个唯一的编号，称为系统调用号。部分编号如下所示：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/asm-i386/unistd.h 8 21 ====================

```
#define __NR_exit          1
#define __NR_fork          2
#define __NR_read          3
#define __NR_write         4
#define __NR_open          5
#define __NR_close         6
#define __NR_waitpid            7
#define __NR_creat         8
#define __NR_link          9
#define __NR_unlink            10
#define __NR_execve            11
#define __NR_chdir        12
#define __NR_time         13
#define __NR_mknod        14
```

系统调用的跳转表是一个函数指针数组，跳转时以系统调用号为下标在数组中找到相应的函数指针。该数组是在 arch/i386/kernel/entry.S 中定义的。数组的大小由常数 NR_syscalls 决定，该常数在 include/linux/sys.h 中定义为 256。目前 Linux 共定义了 221 个系统调用，其余的 30 余项可供用户行添加。数组中对凡是没有定义的下标（系统调用号）都放上一个函数指针，指向 sys_ni_syscall()，其代码在 kernel/sys.c 中：

==================== kernel/sys.c 169 172 ====================

```
asmlinkage long sys_ni_syscall(void)
{
     return -ENOSYS;
}
```

下面即 arch/i386/kernel/entry.S 中数组 sys_call_table 的汇编代码。第 656 行处的 rept NR_syscalls-221 系 gcc 预处理命令。文件经须处理后就会将后面的 657 行重复(NR_syscalls-221)次，也即 35 次。

==================== arch/i386/kernel/entry.S 425 658 ====================

```
425  ENTRY(sys_call_table)
426       .long SYMBOL_NAME(sys_ni_syscall)    /* 0  -  old "setup()" system call*/
427       .long SYMBOL_NAME(sys_exit)
428       .long SYMBOL_NAME(sys_fork)
429       .long SYMBOL_NAME(sys_read)
430       .long SYMBOL_NAME(sys_write)
431       .long SYMBOL_NAME(sys_open)      /* 5 */
432       .long SYMBOL_NAME(sys_close)
433       .long SYMBOL_NAME(sys_waitpid)
434       .long SYMBOL_NAME(sys_creat)
435       .long SYMBOL_NAME(sys_link)
436       .long SYMBOL_NAME(sys_unlink)        /* 10 */
437       .long SYMBOL_NAME(sys_execve)
438       .long SYMBOL_NAME(sys_chdir)
439       .long SYMBOL_NAME(sys_time)
440       .long SYMBOL_NAME(sys_mknod)
```

```
	.long SYMBOL_NAME(sys_chmod)		/* 15 */
	.long SYMBOL_NAME(sys_lchown16)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old break syscall holder */
	.long SYMBOL_NAME(sys_stat)
	.long SYMBOL_NAME(sys_lseek)
	.long SYMBOL_NAME(sys_getpid)		/* 20 */
	.long SYMBOL_NAME(sys_mount)
	.long SYMBOL_NAME(sys_oldumount)
	.long SYMBOL_NAME(sys_setuid16)
	.long SYMBOL_NAME(sys_getuid16)
	.long SYMBOL_NAME(sys_stime)		/* 25 */
	.long SYMBOL_NAME(sys_ptrace)
	.long SYMBOL_NAME(sys_alarm)
	.long SYMBOL_NAME(sys_fstat)
	.long SYMBOL_NAME(sys_pause)
	.long SYMBOL_NAME(sys_utime)		/* 30 */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old stty syscall holder */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old gtty syscall holder */
	.long SYMBOL_NAME(sys_access)
	.long SYMBOL_NAME(sys_nice)
	.long SYMBOL_NAME(sys_ni_syscall)	/* 35 */		/* old ftime syscall holder */
	.long SYMBOL_NAME(sys_sync)
	.long SYMBOL_NAME(sys_kill)
	.long SYMBOL_NAME(sys_rename)
	.long SYMBOL_NAME(sys_mkdir)
	.long SYMBOL_NAME(sys_rmdir)		/* 40 */
	.long SYMBOL_NAME(sys_dup)
	.long SYMBOL_NAME(sys_pipe)
	.long SYMBOL_NAME(sys_times)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old prof syscall holder */
	.long SYMBOL_NAME(sys_brk)		/* 45 */
	.long SYMBOL_NAME(sys_setgid16)
	.long SYMBOL_NAME(sys_getgid16)
	.long SYMBOL_NAME(sys_signal)
	.long SYMBOL_NAME(sys_geteuid16)
	.long SYMBOL_NAME(sys_getegid16) /* 50 */
	.long SYMBOL_NAME(sys_acct)
	.long SYMBOL_NAME(sys_umount)					/* recycled never used phys() */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old lock syscall holder */
	.long SYMBOL_NAME(sys_ioctl)
	.long SYMBOL_NAME(sys_fcntl)		/* 55 */
	.long SYMBOL_NAME(sys_ni_syscall)				/* old mpx syscall holder */
	.long SYMBOL_NAME(sys_setpgid)
	.long SYMBOL_NAME(sys_ni_syscall)				/* old ulimit syscall holder */
	.long SYMBOL_NAME(sys_olduname)
	.long SYMBOL_NAME(sys_umask)		/* 60 */
	.long SYMBOL_NAME(sys_chroot)
	.long SYMBOL_NAME(sys_ustat)
	.long SYMBOL_NAME(sys_dup2)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	.long SYMBOL_NAME(sys_getppid)
	.long SYMBOL_NAME(sys_getpgrp)		/* 65 */
	.long SYMBOL_NAME(sys_setsid)
	.long SYMBOL_NAME(sys_sigaction)
	.long SYMBOL_NAME(sys_sgetmask)
	.long SYMBOL_NAME(sys_ssetmask)
	.long SYMBOL_NAME(sys_setreuid16)	/* 70 */
	.long SYMBOL_NAME(sys_setregid16)
	.long SYMBOL_NAME(sys_sigsuspend)
	.long SYMBOL_NAME(sys_sigpending)
	.long SYMBOL_NAME(sys_sethostname)
	.long SYMBOL_NAME(sys_setrlimit) /* 75 */
	.long SYMBOL_NAME(sys_old_getrlimit)
	.long SYMBOL_NAME(sys_getrusage)
	.long SYMBOL_NAME(sys_gettimeofday)
	.long SYMBOL_NAME(sys_settimeofday)
	.long SYMBOL_NAME(sys_getgroups16)	/* 80 */
	.long SYMBOL_NAME(sys_setgroups16)
	.long SYMBOL_NAME(old_select)
	.long SYMBOL_NAME(sys_symlink)
	.long SYMBOL_NAME(sys_lstat)
	.long SYMBOL_NAME(sys_readlink)		/* 85 */
	.long SYMBOL_NAME(sys_uselib)
	.long SYMBOL_NAME(sys_swapon)
	.long SYMBOL_NAME(sys_reboot)
	.long SYMBOL_NAME(old_readdir)
	.long SYMBOL_NAME(old_mmap)	/* 90 */
	.long SYMBOL_NAME(sys_munmap)
	.long SYMBOL_NAME(sys_truncate)
	.long SYMBOL_NAME(sys_ftruncate)
	.long SYMBOL_NAME(sys_fchmod)
	.long SYMBOL_NAME(sys_fchown16)		/* 95 */
	.long SYMBOL_NAME(sys_getpriority)
	.long SYMBOL_NAME(sys_setpriority)
	.long SYMBOL_NAME(sys_ni_syscall)			/* old profil syscall holder */
	.long SYMBOL_NAME(sys_statfs)
	.long SYMBOL_NAME(sys_fstatfs)		/* 100 */
	.long SYMBOL_NAME(sys_ioperm)
	.long SYMBOL_NAME(sys_socketcall)
	.long SYMBOL_NAME(sys_syslog)
	.long SYMBOL_NAME(sys_setitimer)
	.long SYMBOL_NAME(sys_getitimer) /* 105 */
	.long SYMBOL_NAME(sys_newstat)
	.long SYMBOL_NAME(sys_newlstat)
	.long SYMBOL_NAME(sys_newfstat)
	.long SYMBOL_NAME(sys_uname)
	.long SYMBOL_NAME(sys_iopl)	/* 110 */
	.long SYMBOL_NAME(sys_vhangup)
	.long SYMBOL_NAME(sys_ni_syscall)	/* old "idle" system call */
```

```
	.long SYMBOL_NAME(sys_vm86old)
	.long SYMBOL_NAME(sys_wait4)
	.long SYMBOL_NAME(sys_swapoff)		/* 115 */
	.long SYMBOL_NAME(sys_sysinfo)
	.long SYMBOL_NAME(sys_ipc)
	.long SYMBOL_NAME(sys_fsync)
	.long SYMBOL_NAME(sys_sigreturn)
	.long SYMBOL_NAME(sys_clone)		/* 120 */
	.long SYMBOL_NAME(sys_setdomainname)
	.long SYMBOL_NAME(sys_newuname)
	.long SYMBOL_NAME(sys_modify_ldt)
	.long SYMBOL_NAME(sys_adjtimex)
	.long SYMBOL_NAME(sys_mprotect)		/* 125 */
	.long SYMBOL_NAME(sys_sigprocmask)
	.long SYMBOL_NAME(sys_create_module)
	.long SYMBOL_NAME(sys_init_module)
	.long SYMBOL_NAME(sys_delete_module)
	.long SYMBOL_NAME(sys_get_kernel_syms)	/* 130 */
	.long SYMBOL_NAME(sys_quotactl)
	.long SYMBOL_NAME(sys_getpgid)
	.long SYMBOL_NAME(sys_fchdir)
	.long SYMBOL_NAME(sys_bdflush)
	.long SYMBOL_NAME(sys_sysfs)		/* 135 */
	.long SYMBOL_NAME(sys_personality)
	.long SYMBOL_NAME(sys_ni_syscall)	/* for afs_syscall */
	.long SYMBOL_NAME(sys_setfsuid16)
	.long SYMBOL_NAME(sys_setfsgid16)
	.long SYMBOL_NAME(sys_llseek)		/* 140 */
	.long SYMBOL_NAME(sys_getdents)
	.long SYMBOL_NAME(sys_select)
	.long SYMBOL_NAME(sys_flock)
	.long SYMBOL_NAME(sys_msync)
	.long SYMBOL_NAME(sys_readv)		/* 145 */
	.long SYMBOL_NAME(sys_writev)
	.long SYMBOL_NAME(sys_getsid)
	.long SYMBOL_NAME(sys_fdatasync)
	.long SYMBOL_NAME(sys_sysctl)
	.long SYMBOL_NAME(sys_mlock)		/* 150 */
	.long SYMBOL_NAME(sys_munlock)
	.long SYMBOL_NAME(sys_mlockall)
	.long SYMBOL_NAME(sys_munlockall)
	.long SYMBOL_NAME(sys_sched_setparam)
	.long SYMBOL_NAME(sys_sched_getparam)   /* 155 */
	.long SYMBOL_NAME(sys_sched_setscheduler)
	.long SYMBOL_NAME(sys_sched_getscheduler)
	.long SYMBOL_NAME(sys_sched_yield)
	.long SYMBOL_NAME(sys_sched_get_priority_max)
	.long SYMBOL_NAME(sys_sched_get_priority_min)  /* 160 */
	.long SYMBOL_NAME(sys_sched_rr_get_interval)
```

```
	.long SYMBOL_NAME(sys_nanosleep)
	.long SYMBOL_NAME(sys_mremap)
	.long SYMBOL_NAME(sys_setresuid16)
	.long SYMBOL_NAME(sys_getresuid16)	/* 165 */
	.long SYMBOL_NAME(sys_vm86)
	.long SYMBOL_NAME(sys_query_module)
	.long SYMBOL_NAME(sys_poll)
	.long SYMBOL_NAME(sys_nfsservctl)
	.long SYMBOL_NAME(sys_setresgid16)	/* 170 */
	.long SYMBOL_NAME(sys_getresgid16)
	.long SYMBOL_NAME(sys_prctl)
	.long SYMBOL_NAME(sys_rt_sigreturn)
	.long SYMBOL_NAME(sys_rt_sigaction)
	.long SYMBOL_NAME(sys_rt_sigprocmask)/* 175 */
	.long SYMBOL_NAME(sys_rt_sigpending)
	.long SYMBOL_NAME(sys_rt_sigtimedwait)
	.long SYMBOL_NAME(sys_rt_sigqueueinfo)
	.long SYMBOL_NAME(sys_rt_sigsuspend)
	.long SYMBOL_NAME(sys_pread)		/* 180 */
	.long SYMBOL_NAME(sys_pwrite)
	.long SYMBOL_NAME(sys_chown16)
	.long SYMBOL_NAME(sys_getcwd)
	.long SYMBOL_NAME(sys_capget)
	.long SYMBOL_NAME(sys_capset)		/* 185 */
	.long SYMBOL_NAME(sys_sigaltstack)
	.long SYMBOL_NAME(sys_sendfile)
	.long SYMBOL_NAME(sys_ni_syscall)		/* streams1 */
	.long SYMBOL_NAME(sys_ni_syscall)		/* streams2 */
	.long SYMBOL_NAME(sys_vfork)		/* 190 */
	.long SYMBOL_NAME(sys_getrlimit)
	.long SYMBOL_NAME(sys_mmap2)
	.long SYMBOL_NAME(sys_truncate64)
	.long SYMBOL_NAME(sys_ftruncate64)
	.long SYMBOL_NAME(sys_stat64)		/* 195 */
	.long SYMBOL_NAME(sys_lstat64)
	.long SYMBOL_NAME(sys_fstat64)
	.long SYMBOL_NAME(sys_lchown)
	.long SYMBOL_NAME(sys_getuid)
	.long SYMBOL_NAME(sys_getgid)		/* 200 */
	.long SYMBOL_NAME(sys_geteuid)
	.long SYMBOL_NAME(sys_getegid)
	.long SYMBOL_NAME(sys_setreuid)
	.long SYMBOL_NAME(sys_setregid)
	.long SYMBOL_NAME(sys_getgroups) /* 205 */
	.long SYMBOL_NAME(sys_setgroups)
	.long SYMBOL_NAME(sys_fchown)
	.long SYMBOL_NAME(sys_setresuid)
	.long SYMBOL_NAME(sys_getresuid)
	.long SYMBOL_NAME(sys_setresgid) /* 210 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	.long SYMBOL_NAME(sys_getresgid)
	.long SYMBOL_NAME(sys_chown)
	.long SYMBOL_NAME(sys_setuid)
	.long SYMBOL_NAME(sys_setgid)
	.long SYMBOL_NAME(sys_setfsuid)		/* 215 */
	.long SYMBOL_NAME(sys_setfsgid)
	.long SYMBOL_NAME(sys_pivot_root)
	.long SYMBOL_NAME(sys_mincore)
	.long SYMBOL_NAME(sys_madvise)
	.long SYMBOL_NAME(sys_getdents64)	/* 220 */
	.long SYMBOL_NAME(sys_fcntl64)
	.long SYMBOL_NAME(sys_ni_syscall)	/* reserved for TUX */

	/*
	 * NOTE!! This doesn't have to be exact - we just have
	 * to make sure we have _enough_ of the "sys_ni_syscall"
	 * entries. Don't panic if you notice that this hasn't
	 * been shrunk every time we add a new system call.
	 */
	.rept NR_syscalls-221
		.long SYMBOL_NAME(sys_ni_syscall)
	.endr
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第4章 进程与进程调度

4.1 进程四要素

要给“进程”下一个确切的定义不是件容易的事。不过，一般来说 Linux 系统中的进程都具备下列诸要素：

(1) 有一段程序供其执行，就好像一场戏要有个剧本一样。这段程序不一定是进程所专有，可以与其它进程共用，就好像不同剧团的许多场演出可以共用一个剧本一样。

(2) 有起码的“私有财产”，这就是进程专用的系统堆栈空间。

(3) 有“户口”，这就是在内核中的一个 task_struct 数据结构，操作系统教科书中常称为“进程控制块”。有了这个数据结构，进程才能成为内核调度的一个基本单位接受内核的调度。同时，这个结构又是进程的“财产登记卡”，记录着进程所占用的各项资源。

(4) 有独立的存储空间，意味着拥有专有的用户空间；进一步，还意味着除前述的系统空间堆栈外还有其专用的用户空间堆栈。注意，系统空间是不能独立的，任何进程都不可能直接（不通过系统调用）改变系统空间的内容（除其本身的系统空间堆栈以外）。

这四条都是必要条件，缺了其中任何一条就不称其为“进程”。如果只具备了前面二条而缺第四条，那就称为“线程”。特别地，如果完全没有用户空间，就称为“内核线程”（kernel thread）；而如果共享用户空间则就称为“用户线程”。在不致引起混淆的场合，二者也都往往简称为“线程”。读者在第 2 章中看到过的 kswapd，就是一个内核线程。读者要注意，不要把这里的“线程”与有些系统中在用户空间的同一进程内实现的“线程”相混淆。那种线程显然不拥有独立、专用的系统堆栈，也不作为一个调度单位直接受内核调度。而且，既然 Linux 内核提供了对线程的支持，一般也就没有必要再在进程内部，即用户空间中自行实现线程。

另一方面，进程与线程的区分也不是十分严格的，一般在讲到进程时常常也包括了线程。事实上，在 Linux（以及 Unix）系统中，许多进程在“诞生”之初都与其父进程共用同一个存储空间，所以严格说来还是线程；但是子进程可以建立其自己的存储空间，并与父进程分道扬镳，成为真正意义上的进程。再说，线程也有“ pid”，也有 task_struct 结构，所以这两个词在使用中有时候并不严格加以区分，要根据上下文理解其含意。

还有，在 Linux 系统中“进程”（process）和“任务”（task）是同一个意思，在内核的代码中也常常混用这两个名词和概念。例如，每一个进程都要有一个 task_struct 数据结构，而其号码却又是 pid；唤醒一个睡眠进程的函数名为 wake_up_process()。之所以有这样的情况是因为 Linux 源自 Unix 和 i386 系统结构，而 Unix 中的进程在 Intel 的技术资料中则称为“任务”（严格说来有点区别，但是对 Linux 和 Unix 的实现来说是一码事）。

Linux 系统运行时的第一个进程是在初始化阶段“捏造”出来的。而此后的进程或线段则都是由一个业已存在的进程像细胞分裂那样通过系统调用复制出来的，称为“ fork”（分叉）或“ clone”（克隆）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

除上述最起码的“财产”，即 task_struct 数据结构和系统堆栈之外，一个进程还要有些附加的资源。例如，上面说过，“独立”的存储空间意味着进程拥有用户空间，因此就要有用于虚存管理的 mm_struct 数据结构以及下属的 vm_area 数据结构，以及相应的页面目录项和页面表。但那些都是第二位的，从属于 task_struct 的资源，而 task_struct 数据结构则在这方面起着登记卡的作用。至于进程的具体实现，则在相当程度上取决了宿主 CPU 的系统结构。

在转入详细介绍进程的各个要素之前，我们先讲一下 i386 系统结构所提供的进程管理机制以及 Linux 内核对这种机制的特殊运用和处理。读者可以结合第 2 章中的有关内容阅读。

Intel 在 i386 系统结构的设计中考虑到了进程（任务）的管理和调度，并从硬件上支持任务间的切换。为此目的，Intel 在 i386 系统结构中增设了另一种新的段，叫做“任务状态段 ” TSS。一个 TSS 虽说像代码段、数据段等一样，也是一个“段”，实际上却只是一个 104 字节的数据结构、或曰控制块，用以记录一个任务的关键性的状态信息，包括：

- 任务切换前夕（也就是切入点上）该任务各通用寄存器的内容。
- 任务切换前夕（切入点上）该任务各个段寄存器（包括 ES、CS、SS、DS、FS 和 GS）的内容。
- 在务切换前夕（切入点上）该任务 EFLAGS 寄存器的内容。
- 任务切换前夕（切入点上）该任务指令地址寄存器 EIP 的内容。
- 指向前一个任务的 TSS 结构的段选择码。当前任务执行 IRET 指令时，就返回到由这个段选择码所指的（TSS 所代表的）任务（返回地址则由堆栈决定）。
- 该任务的 LDT 段选择码，它指向任务的 LDT。
- 控制寄存器 CR3 的内容，它指向任务的页面目录。
- 三个堆栈指针，分别为当任务运行于 0 级、1 级和 2 级时的堆栈指针，包括堆栈段寄存器 SS0、SS1 和 SS2，以及 ESP0、ESP1 和 ESP2 的内容。注意，在 CPU 中只有一个 SS 和一个 ESP 寄存器，但是 CPU 在进入新的运行级别时会自动从当前任务的 TSS 中装入相应 SS 和 ESP 的内容，实现堆栈的切换。
- 一个用于程序跟踪的标志位 T。当 T 标志位为 1 时，CPU 就会在切入该进程时产生一次 debug 异常，这样就可以在 debug 异常的服务程序中安排所需的操作，如加以记录、显示、等等。
- 在一个 TSS 段中，除了基本的 104 字节的 TSS 结构以外，还可以有一些附加的信息。其中之一是表示 I/O 权限的位图。i386 系统结构允许 I/O 指令在比 0 级低的状态下执行，也就是说可以将外设驱动实现于一个既非内核（0 级）也非用户（3 级）的空间中，这个位图就是用于这个目的。另一个是“中断重定向位图”，用于 vm86 模式。

像其它的“段”一样，TSS 也要在段描述表中有个表项。不过 TSS 的描述项只能在 GDT 中，而不能放在任何一个 LDT 中或 IDT 中。如果通过一个段选择项访问一个 TSS，而选择项中的 TI 标志位为 1（表示使用 LDT），就会产生一次“总保护 ” GP 异常。TSS 描述项的结构与其它的段描述项基本相同（参考第 2 章），但有一个 B（Busy）标志位，表示相应 TSS 所代表的任务是否正在运行或者正被中断。

另外，CPU 中还增设了一个“任务寄存器 ” TR，指向当前任务的 TSS。相应地，还增加了一条指令 LTR，对 TR 寄存器进行装入操作。像 CS 和 DS 一样，TR 也有一个不可见的部分，每当将一个段选择码装入到 TR 中时，CPU 就自动找到所选择的 TSS 描述项并将其装入到 TR 中的不可见部分，以加速以后对该 TSS 段的访问。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

还有，在 IDT 表中，除中断门、陷阱门和调用门外，还定义了一种“任务门”。任务门中包含有一个 TSS 段选择码。当 CPU 因中断而穿过一个任务门时，就会将任务门中的段选择码自动装入 TR，使 TR 指向新的 TSS，并完成任务切换。CPU 还可以通过 JMP 和 CALL 指令实现任务切换，当跳转或调用的目标段（代码段）实际上指向 GDT 表中的一个 TSS 描述项时，就会引起一次任务切换。

Intel 的这种设计确实很周到，也为任务切换提供了一个非常简洁的机制。但是，请读者注意，由 CPU 自动完成的这种任务切换并不是像读者可能误以为的那样只相当于“一条指令”。实际下，i386 的系统结构基本上是 CISC 的，而通过 JMP 指令或 CALL 指令（或中断）完成任务切换的过程可以说是典型的、甚至是极端的“复杂指令”执行过程，其执行过程长达 300 多个 CPU 时钟周期（一条 POP 指令占 12 个 CPU 时钟周期）。在执行的过程中，CPU 实际上做了所有可能需要做的事，而其中有的事在一定的条件下本来是可以简化的，有的事则可能在一定的条件下应该按不同的方式组合。所以，i386 CPU 所提供的这种任务切换机制就好像是一种“高级语言”的成分。你固然可以用它，但对于操作系统的设计和实现而言，你往往会选择“汇编语言”来实现这个机制，以达到更高的效率和更大的灵活性。更重要的是，任务的切换往往不是孤立的，常常跟其它的操作联系在一起。例如，在 Unix 和 Linux 系统中，任务切换就只发生于系统空间，因而与系统调用和中断密切联系在一起，并且有许多操作可以合并。

就如对 i386 所提供的许多其它功能一样，读者将会看到，Linux 内核实际上并不使用 i386 CPU 硬件提供的任务切换机制。不过，由于 i386 CPU 要求软件设置 TR 及 TSS，内核中便只好“走过场”地设置好 TR 及 TSS 以满足 CPU 的要求。但是，内核中并不使用任务门、也不允许使用 JMP 或 CALL 指令实施任务切换。内核只是在初始化阶段设置 TR，使之指向一个 TSS，从此以后就再不改变 TR 的内容了。也就是说，每个 CPU（如果有多个 CPU 的话）在初始化以后的全部运行过程中永远各自使用同一个 TSS。同时，内核也不依靠 TSS 保存每个进程切换时的寄存器副本，而是将这些寄存器的副本保存在各个进程自己的系统空间堆栈中，就如读者在第 3 章中所看到的那样。

这样一来，TSS 中的绝大部分内容已经失去了原来的意义。可是，在第 3 章中讲过，当 CPU 因中断或系统调用而从用户空间进入系统空间时，会由于运行级别的变化而自动更换堆栈。而新的堆栈指针，包括堆栈段寄存器 SS 的内容和堆栈指引寄存器 ESP 的内容，则取自“当前”任务的 TSS。由于在 Linux 中只使用两个运行级别，即 0 级和 3 级，所以 TSS 中为另两个级别（即 1 级和 2 级）设置的堆栈指针副本也失去了意义。于是，对于 Linux 内核来说，TSS 中有意义的就只剩下了 0 级的堆栈指针，也就是 SS0 和 ESP0 两项了。Intel 原来的意图是让 TR 的内容，随着不同的 TSS，随着任务的切换而走马灯似地转。可是在 Linux 内核中却变成了“铁打的营盘流水的兵”：就一个 TSS，像一座营盘，一经建立就再也不动了。而里面的内容，也就是当前任务的系统堆栈指针，则随着进程的调度切换而流水似地变动。这里的原因在于：改变 TSS 中 SS0 和 ESP0 所花的开销比通过装入 TR 以更换一个 TSS 要小得多。因此，在 Linux 内核中，TSS 并不是属于某个进程的资源，而是个全局性的公共资源。在多处理器的情况下，尽管内核中确实有多个 TSS，但是每个 CPU 仍旧只有一个 TSS，一经装入就不再变了。

那么，这个 TSS 是什么样的呢？请看 include/asm-i386/processor.h 中对 INIT_TSS 的定义：

==================== include/asm-i386/processor.h 392 406 ====================

```
#define INIT_TSS  {                                     \
        0,0, /* back_link, __blh */                     \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	sizeof(init_stack) + (long) &init_stack, /* esp0 */\
	__KERNEL_DS, 0, /* ss0 */                  \
	0,0,0,0,0,0, /* stack1, stack2 */          \
	0, /* cr3 */                               \
	0,0, /* eip,eflags */                      \
	0,0,0,0, /* eax,ecx,edx,ebx */             \
	0,0,0,0, /* esp,ebp,esi,edi */             \
	0,0,0,0,0,0, /* es,cs,ss */                \
	0,0,0,0,0,0, /* ds,fs,gs */                \
	__LDT(0),0, /* ldt */                      \
	0, INVALID_IO_BITMAP_OFFSET, /* tace, bitmap */          \
	{~0, } /* ioperm */                        \
}
```

这里把系统中第一个进程的 SS0 设置成__KERNEL_DS，而把 ESP0 设置成指向&init_stack 的顶端。init_stack 的定义如下：

==================== include/asm-i386/processor.h 452 452 ====================

```
#define init_stack (init_task_union.stack)
```

==================== arch/i386/kernel/init_task.c 22 24 ====================

```
union task_union init_task_union
	__attribute__((__section__(".data.init_task"))) =
		{ INIT_TASK(init_task_union.task) };
```

==================== include/linux/sched.h 484 487 ====================

```
union task_union {
	struct task_struct task;
	unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
```

==================== include/linux/sched.h 480 482 ====================

```
#ifndef INIT_TASK_SIZE
# define INIT_TASK_SIZE 2048*sizeof(long)
#endif
```

对 INIT_TSS 的引用则在 arch/i386/kernel/init_task.c 中给出：

==================== arch/i386/kernel/init_task.c 26 33 ====================

```
/*
 * per-CPU TSS segments. Threads are completely 'soft' on Linux,
 * no more per-task TSS's. The TSS size is kept cacheline-aligned
 * so they are allowed to end up in the .data.cacheline_aligned
 * section. Since TSS's are completely CPU-local, we want them
 * on exact cacheline boundaries, to eliminate cacheline ping-pong.
 */
struct tss_struct init_tss[NR_CPUS] __cacheline_aligned = { [0 ... NR_CPUS-1] = INIT_TSS };
```

结构数组 init_tss 的大小为 NR_CPUS，即系统中 CPU 的个数。每个 TSS 的内容都相同，都由 INIT_TSS 定义。此外，每个 TSS 的起始地址都与高速缓存中的缓冲行对齐。

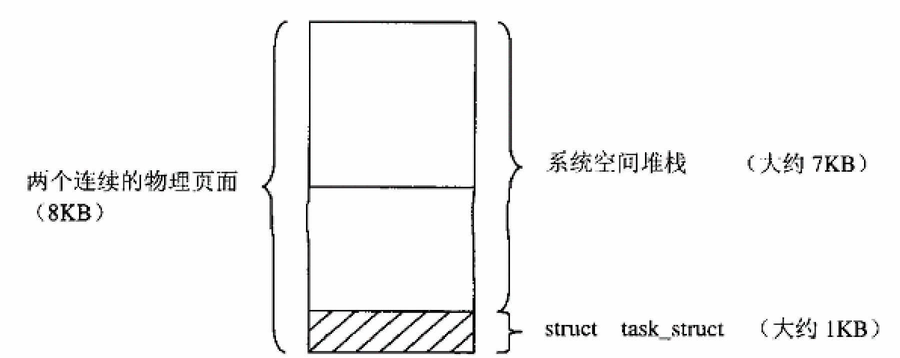
数据结构 tss_struct 是在 include/asm-i386/processor.h 中定义的，它反映了 TSS 段的结构：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/asm-i386/processor.h 327 356 ====================

```
struct tss_struct {
	unsigned short	back_link,__blh;
	unsigned long esp0;
	unsigned short	ss0,__ss0h;
	unsigned long esp1;
	unsigned short	ss1,__ss1h;
	unsigned long esp2;
	unsigned short	ss2,__ss2h;
	unsigned long __cr3;
	unsigned long eip;
	unsigned long eflags;
	unsigned long eax,ecx,edx,ebx;
	unsigned long esp;
	unsigned long ebp;
	unsigned long esi;
	unsigned long edi;
	unsigned short	es, __esh;
	unsigned short	cs, __csh;
	unsigned short	ss, __ssh;
	unsigned short	ds, __dsh;
	unsigned short	fs, __fsh;
	unsigned short	gs, __gsh;
	unsigned short	ldt, __ldth;
	unsigned short	trace, bitmap;
	unsigned long io_bitmap[IO_BITMAP_SIZE+1];
	/*
	 * pads the TSS to be cacheline-aligned (size is 0x100)
	 */
	unsigned long __cacheline_filler[5];
};
```

前面讲过，每个进程都有一个 task_struct 数据结构和一片用作系统空间堆栈的存储空间。这二者缺一不可，又有紧密的联系，所以在物理存储空间中也连在一起。内核在为每个进程分配一个 task_struct 结构时，实际上分配两个连续的物理页面（共 8192 字节）。这两个页面的底部用作进程的 task_struct 结构；而在结构的上面就用作进程的系统空间堆栈，见图 4.1。



PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

图 4.1　进程系统堆栈示意图

数据结构 task_struct 的大小约 1K 字节，所以进程系统空间堆栈的大小约为 7K 字节。注意，系统空间堆栈的空间不像用户空间堆栈那样可以在运行时动态地扩展（见第 2 章），而是静态地确定了的。所以，在中断服务程序、内核软中断服务程序以及其它设备驱动程序的设计中，应注意不能让这些函数嵌套太深，同时，在这些函数中也不宜使用太多、太大的局部变量。像下面程序中这样的局部变量就应该避免：

```
int something()
{
char buf[1024];
......
}
```

这里的 buf 是局部变量，因为是在堆栈中，它一下子就耗去了 1K 字节，显然是不合适的。

进程 task_struct 结构以及系统空间堆栈的这种特殊安排，决定了内核中一些宏操作的定义（include/asm-i386/processor.h）：

==================== include/asm-i386/processor.h 446 448 ====================

```
#define THREAD_SIZE (2*PAGE_SIZE)
#define alloc_task_struct() ((struct task_struct *) __get_free_pages(GFP_KERNEL,1))
#define free_task_struct(p) free_pages((unsigned long) (p), 1)
```

THREAD_SIZE 定义为两个页面，表示每个内核线程（一个进程必定同时又是一个内核线程）的这两项基本资源所占的物理存储空间大小。至于 alloc_task_struct()的实现，读者也许会想像成这样：

```
struct task_struct *t = kmalloc(sizeof(struct task_struct));
```

实际上却不是，这是因为所分配的并不仅仅是 task_struct 数据结构的大小，而是连同系统空间堆栈所需的空间一起分配。注意__get_free_pages()中第二个参数的值 1 表示 2^1，也就是两个页面。

当进程在系统空间运行时，常常需要访问当前进程自身的 task_struct 数据结构。为此目的，内核中（include/asm-i386/current.h）定义了一个宏操作 current，提供指向当前进程 task_struct 结构的指针：

==================== include/asm-i386/current.h 6 13 ====================

```
6    static inline struct task_struct * get_current(void)
7    {
8         struct task_struct *current;
9         __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
10        return current;
11    }
12
13   #define current get_current()
```

第 9 行通过将当前的堆栈指针寄存器 ESP 的内容与 8191UL（0xffffe00）相“与”而得到当前进程 task_struct 结构的起始地址（汇编代码的解释可参看第 2 章和第 3 章的几个例子）。结合前面的图 4.1 和说明，读者应不难理解为什么这样就可以得到所需的地址。

那么，为什么不把这地址放在一个全局量中，使得每次调度一个新的进程运行时就将该进程的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

task_struct 结构的起始地址写入这个变量，以后便随时可用，这样不是更有效吗？答案恰恰相反。一条 AND 指令的执行只需 4 个 CPU 时钟周期，而一条从寄存器到寄存器的 MOV 指令也才 2 个 CPU 时钟周期，所以，像这样在需要时才临时把它计算出来反而效率更高。读者从这里也可以看出，高水平的系统程序员的“抠门”真是到了极点。

与此相类似的，还有在进入中断和系统调用时所引用的宏操作 GET_CURRENT，那是在 include/asm-i386/hw_irq.h 中定义的：

==================== include/asm-i386/hw_irq.h 113 115 ====================

```
#define GET_CURRENT \
	"movl %esp, %ebx\n\t" \
	"andl $-8192, %ebx\n\t"
```

我们在第 2 章中跳过了对这段程序的解释，同为那时还没有讲到进程的系统空间堆栈与其 task_struct 结构之间的关系。

task_struct 的定义在 include/linux/sched.h 中给出：

==================== include/linux/sched.h 277 397 ====================

```
struct task_struct {
	/*
	 * offsets of these are hardcoded elsewhere - touch with care
	 */
	volatile long state;	/* -1 unrunnable, 0 runnable, >0 stopped */
	unsigned long flags;	/* per process flags, defined below */
	int sigpending;
	mm_segment_t addr_limit;	/* thread address space:
					0-0xBFFFFFFF for user-thead
					0-0xFFFFFFFF for kernel-thread
				 */
	struct exec_domain *exec_domain;
	volatile long need_resched;
	unsigned long ptrace;

	int lock_depth;		/* Lock depth */

/*
 * offset 32 begins here on 32-bit platforms. We keep
 * all fields in a single cacheline that are needed for
 * the goodness() loop in schedule().
 */
	long counter;
	long nice;
	unsigned long policy;
	struct mm_struct *mm;
	int has_cpu, processor;
	unsigned long cpus_allowed;
	/*
	 * (only the 'next' pointer fits into the cacheline, but
```

```
	 * that's just fine.)
	 */
	struct list_head run_list;
	unsigned long sleep_time;

	struct task_struct *next_task, *prev_task;
	struct mm_struct *active_mm;

/* task state */
	struct linux_binfmt *binfmt;
	int exit_code, exit_signal;
	int pdeath_signal;  /*  The signal sent when the parent dies  */
	/* ??? */
	unsigned long personality;
	int dumpable:1;
	int did_exec:1;
	pid_t pid;
	pid_t pgrp;
	pid_t tty_old_pgrp;
	pid_t session;
	pid_t tgid;
	/* boolean value for session group leader */
	int leader;
	/*
	 * pointers to (original) parent process, youngest child, younger sibling,
	 * older sibling, respectively.  (p->father can be replaced with
	 * p->p_pptr->pid)
	 */
	struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
	struct list_head thread_group;

	/* PID hash table linkage. */
	struct task_struct *pidhash_next;
	struct task_struct **pidhash_pprev;

	wait_queue_head_t wait_chldexit;	/* for wait4() */
	struct semaphore *vfork_sem;		/* for vfork() */
	unsigned long rt_priority;
	unsigned long it_real_value, it_prof_value, it_virt_value;
	unsigned long it_real_incr, it_prof_incr, it_virt_incr;
	struct timer_list real_timer;
	struct tms times;
	unsigned long start_time;
	long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
	unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
	int swappable:1;
/* process credentials */
	uid_t uid,euid,suid,fsuid;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
356       gid_t gid,egid,sgid,fsgid;
357       int ngroups;
358       gid_t     groups[NGROUPS];
359       kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
360       int keep_capabilities:1;
361       struct user_struct *user;
362  /* limits */
363       struct rlimit rlim[RLIM_NLIMITS];
364       unsigned short used_math;
365       char comm[16];
366  /* file system info */
367       int link_count;
368       struct tty_struct *tty; /* NULL if no tty */
369       unsigned int locks; /* How many file locks are being held */
370  /* ipc stuff */
371       struct sem_undo *semundo;
372       struct sem_queue *semsleeping;
373  /* CPU-specific state of this task */
374       struct thread_struct thread;
375  /* filesystem information */
376       struct fs_struct *fs;
377  /* open file information */
378       struct files_struct *files;
379  /* signal handlers */
380       spinlock_t sigmask_lock;    /* Protects signal and blocked */
381       struct signal_struct *sig;
382
383       sigset_t blocked;
384       struct sigpending pending;
385
386       unsigned long sas_ss_sp;
387       size_t sas_ss_size;
388       int (*notifier)(void *priv);
389       void *notifier_data;
390       sigset_t *notifier_mask;
391
392  /* Thread group tracking */
393       u32 parent_exec_id;
394       u32 self_exec_id;
395  /* Protection of (de-)allocation: mm, files, fs, tty */
396       spinlock_t alloc_lock;
397  };
```

先把结构中几个特别重要的成分介绍一下，其余则留待以后用到的时候再来介绍。这些成分大体可以分成状态、性质、资源和组织等几大类。

第 281 行的 state 表示进程当前的运行状态，具体定义见 include/linux/sched.h：

===================== include/linux/sched.h 84 88 =====================

```
84   #define TASK_RUNNING        0
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE     2
#define TASK_ZOMBIE          4
#define TASK_STOPPED         8
```

状态 TASK_INTERRUPTIBLE 和 TASK_UNINTERRUPTIBLE 均表示进程处于睡眠状态。但是，TASK_UNINTERRUPTIBLE 表示进程处于“深度睡眠”而不受“信号”（signal，也称“软中断”）的打扰，而 TASK_INTERRUPTIBLE 则可以因“信号”的到来而被唤醒。内核中提供了不同的函数，让一个进程进入不同深度的睡眠或将进程从睡眠中唤醒。具体地说，函数 sleep_on()和 wake_up()用于深度睡眠，而 interruptible_sleep_on()和 wake_up_interruptible()则用于浅度睡眠。深度睡眠一般只用于临界区和关键性的部位，而“可中断”的睡眠那就是通用的了。特别，当进程在“阻塞性”（blocking）的系统调用中等待某一事件发生时，应该进入“可中断”睡眠而不应深度睡眠。例如，当进程等待操作人员按某个键的时候，就不应该进入深度睡眠，否则就不能对别的事件作出反应，别的进程就不能通过发一个信号来“杀”掉这个进程了。还应该注意，这里的 INTERRUPTIBLE 或 UNINTERRUPTIBLE 跟“中断”毫无关系，而只是说睡眠能否因其它事件而中断，即唤醒。不过，所谓其它事件主要是“信号”，而信号的概念实际上与中断的概念是相同的，所以这里所谓 INTERRUPTIBLE 也是指这种“软中断”而言。

TASK_RUNNING 状态并不是表示一个进程正在执行中，或者说这个进程就是“当前进程”，而是表示这个进程可以被调度执行而成为当前进程。当进程处于这样的可执行（或就绪）状态时，内核就将该进程的 task_struct 结构通过其队列头 run_list（见 309 行）挂入一个“运行队列”。

TASK_ZOMBIE 状态表示进程已经“去世”（exit）而“户口”尚未注销。

TASK_STOPPED 主要用于调试目的。进程接收到一个 SIGSTOP 信号后就将运行状态改成 TASK_STOPPED 而进入“挂起”状态，然后在接收到一个 SIGCONT 信号时又恢复继续运行。

在本章“进程的调度与切换”一节中有一个进程的状态转换示意图（第 367 页图 4.4），读者不妨先翻过去看一下。

第 282 行中的 flags 也是反映进程状态的信息，但并不是运行状态，而是与管理有关的其它信息。这些标志位也是在 include/linux/sched.h 中定义的：

==================== include/linux/sched.h 399 413 ====================

```
/*
 * Per process flags
 */
#define PF_ALIGNWARN    0x00000001    /* Print alignment warning msgs */
                        /* Not implemented yet, only for 486*/
#define PF_STARTING     0x00000002    /* being created */
#define PF_EXITING 0x00000004    /* getting shut down */
#define PF_FORKNOEXEC   0x00000040    /* forked but didn't exec */
#define PF_SUPERPRIV    0x00000100    /* used super-user privileges */
#define PF_DUMPCORE     0x00000200    /* dumped core */
#define PF_SIGNALED     0x00000400    /* killed by a signal */
#define PF_MEMALLOC     0x00000800    /* Allocating memory */
#define PF_VFORK   0x00001000    /* Wake up parent in mm_release */

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define PF_USEDFPU 0x00100000    /* task used FPU this quantum (SMP) */
```

代码作者所加的注解已经说明了各个标志位的作用，这里就不多说了。

除上述的 state 和 flags 以外，反映当前状态的成分还有下面这么一些：

sigpending——表示进程收到了“信号”但尚未处理。与这个标志相联系的是与信号队列有关 sigqueue、sigqueue_tail、sig 等指针以及 sigmask_lock、signal、blocked 等成分。请详见“进程间通信”中的“信号”一节，以及本章中的有关叙述。

counter——与调度有关，详见“进程的调度与切换”一节。

need_resched——与调度有关，表示 CPU 从系统空间返回用户空间前夕要进行一次调度。

上述当前状态都反映了进程的动态特征，还有一些则反映静态特征：

addr_limit——虚存地址空间的上限。对进程而言是其用户空间的上限，所以是 0xbfffffff；对内核线程而言则是系统空间的上限，所以是 0xffffffff。

personality——由于 Unix 有许多不同的版本和变种，应用程序也就有了适用范围，例如 Unix SVR4 的应用程序就未必与为 Linux 开发的其它软件完全兼容。所以根据执行程序的不同，每个进程都有其“个性”。文件 include/linux/personality.h 中定义了有关的常数：

==================== include/linux/personality.h 8 32 ====================

```
/* Flags for bug emulation. These occupy the top three bytes. */
#define STICKY_TIMEOUTS      0x4000000
#define WHOLE_SECONDS        0x2000000
#define ADDR_LIMIT_32BIT     0x0800000

/* Personality types. These go in the low byte. Avoid using the top bit,
 * it will conflict with error returns.
 */
#define PER_MASK        (0x00ff)
#define PER_LINUX       (0x0000)
#define PER_LINUX_32BIT      (0x0000 | ADDR_LIMIT_32BIT)
#define PER_SVR4        (0x0001 | STICKY_TIMEOUTS)
#define PER_SVR3        (0x0002 | STICKY_TIMEOUTS)
#define PER_SCOSVR3          (0x0003 | STICKY_TIMEOUTS | WHOLE_SECONDS)
#define PER_WYSEV386         (0x0004 | STICKY_TIMEOUTS)
#define PER_ISCR4       (0x0005 | STICKY_TIMEOUTS)
#define PER_BSD              (0x0006)
#define PER_SUNOS       (PER_BSD | STICKY_TIMEOUTS)
#define PER_XENIX       (0x0007 | STICKY_TIMEOUTS)
#define PER_LINUX32          (0x0008)
#define PER_IRIX32             (0x0009 | STICKY_TIMEOUTS) /* IRIX5 32-bit    */
#define PER_IRIXN32            (0x000a | STICKY_TIMEOUTS) /* IRIX6 new 32-bit */
#define PER_IRIX64             (0x000b | STICKY_TIMEOUTS) /* IRIX6 64-bit    */
#define PER_RISCOS      (0x000c)
#define PER_SOLARIS          (0x000d | STICKY_TIMEOUTS)
```

exec_domain——除了 personality 以外，应用程序还有一些其它的版本间的差异；从而形成了不同

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的“执行域”。这个指针就指向描述本进程所属执行域的数据结构。

binfmt——应用程序的文件格式，如 a.out、elf 等。详见“系统调用 exec()”一节。

exit_code、exit_signal、pdeath_signal——详见“系统调用 exit()与 wait4()”。

pid——进程号。

pgrp、session、leader——当一个用户登录到系统时，就开始了一个进程组（session），此后创建的进程都属于这同一个 session。此外，若干进程可以通过“管道”组合在一起，如“ ls | wc -l”，从而形成进程组。详见“系统调用 exec”一节。

priority、rt_priority——优先级别以及“实时”优先级别，详见“进程的调度与切换”。

policy——适用于本进程的调度政策，详见“进程的调度与切换”。

parent_exec_id、self_exec_id——与进程组（session）有关，见“系统调用 exit()与 wait4()”。

uid、euid、suid、fsuid、gid、egid、sgid、fsgid——主要与文件操作权限有关，见“文件系统”一章。

cap_effective、cap_inheritable、cap_permitted——一般进程都不能“为所欲为”，而是各自被赋予了各种不同的权限。例如，一个进程是否可以通过系统调用 ptrace()跟踪另一个进程，就是由该进程是否具有 CAP_SYS_PTRACE 授权决定的；一个进程是否有权重新引导操作系统，则取决于该进程是否具有 CAP_SYS_BOOT 授权。这样，就把进程的各种权限分细了，而不再是笼统地取决于一个进程是否是“特权用户”进程。文件 include/linux/capability.h 中定义了许多这样的权限，代码的作者还加了相当详细的注解（详见“文件系统”一章）。每一种权限都由一个标志位代表，内核中提供了一个 inline 函数 capable()，用来检验当前进程是否具有某种权限。如 capable(CAP_SYS_BOOT)，就是检查当前进程是否有权重引导操作系统（返回非 0 表示有权）。值得注意的是，对操作权限的这种划分与文件访问权限结合在一起，形成了系统安全性的基础。在现今的网络时代，这种安全性正在变得愈来愈重要，而这方面的研究与发展也是一个重要的课题。

user——指向一个 user_struct 结构，该数据结构代表着进程所属的用户。注意这跟 Unix 内核中每个进程的 user 结构是两码事。Linux 内核中的 user 结构是非常简单的，详见“系统调用 fork()”一节。

rlim——这是一个结构数组，表明进程对各种资源的使用数量所受的限制。读者在“存储管理”一章中已经看到过其应用。数据结构 rlimit 在 include/linux/resource.h 中定义的：

==================== include/linux/resource.h 40 43 ====================

```
struct rlimit {
	unsigned long rlim_cur;
	unsigned long rlim_max;
};
```

对 i386 环境而言，进程可用资源共有 RLIM_NLIMITS 项，即 10 项。每种资源的限制在文件 include/asm-i386/resource.h 中给出：

==================== include/asm-i386/resource.h 4 20 ====================

```
/*
 * Resource limits
 */

#define RLIMIT_CPU 0		/* CPU time in ms */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define RLIMIT_FSIZE    1         /* Maximum filesize */
#define RLIMIT_DATA     2         /* max data size */
#define RLIMIT_STACK    3         /* max stack size */
#define RLIMIT_CORE     4         /* max core file size */
#define RLIMIT_RSS 5          /* max resident set size */
#define RLIMIT_NPROC    6         /* max number of processes */
#define RLIMIT_NOFILE   7         /* max number of open files */
#define RLIMIT_MEMLOCK 8          /* max locked-in-memory address space */
#define RLIMIT_AS  9          /* address space limit */
#define RLIMIT_LOCKS    10        /* maximum file locks held */

#define RLIM_NLIMITS    11
```

还有一些成分代表进程所占有和使用的资源，如 mm、active_mm、fs、files、tty、real_timer、times、it_real_value 等，对这些成分都有专门的章节加以介绍，这里就不重复了。

至于统计信息，则主要有 per_cpu_utime[]和 per_cpu_stime[]两个数组，表示该进程在各个处理器上（在多处理器 SMP 结构中，一个进程可以受调度在不同的处理器上运行）运行于用户空间和系统空间的累计时间。而数据结构 times 中则是对这些时间的汇总。此外，还有对发生页面异常次数的统计 min_flt、maj_flt 以及换入 / 换出的次数 nswap 等。当一个进程通过 do_exit()结束其生命时，该进程的有关统计信息要合并到父进程中，所以对每项统计信息都有一项相应的“总计”信息，如相对于 min_flt 有 cmin_flt，在数据结构 times 中相对于 tms_utime 有 tms_cutime 等。

最后，每一个进程都不是孤立地存在于系统中，而总是根据不同的目的、关系和需要与其它的进程相联系。从内核的角度看，则是要按不同的目的和性质将每个进程纳入不同的组织中。第一个组织是由每个进程的“家庭与社会关系”形成的“宗族”或“家谱”。这是一种树型的组织，通过指针 p_opptr、p_pptr、p_cptr、p_ysptr 和 p_osptr 构成。其中 p_opptr 和 p_pptr 指向父进程的 task_struct 结构，p_cptr 指向最“年轻”的子进程，而 p_ysptr 和 p_osptr 则分别指向其“弟弟”和“哥哥”，从而形成一个子进程链。这些指针确定了一个进程在其“宗族”中的上、下、左、右关系，详见本章中对 fork()和 exit()的叙述。

图 4.2 就是这个进程“家谱”的示意图。

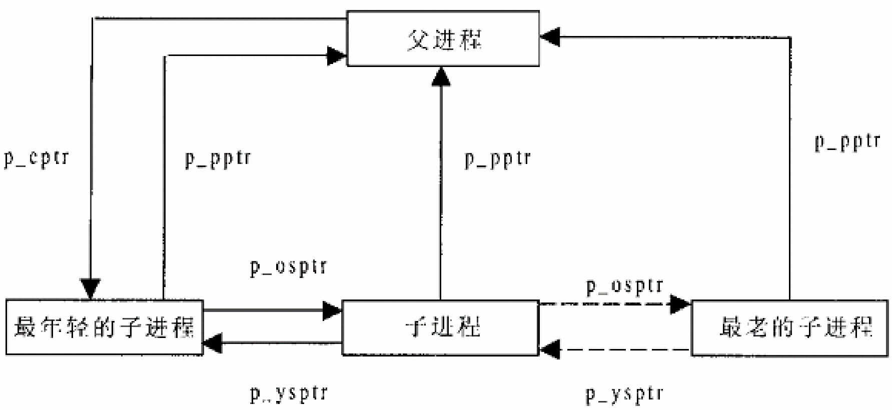


图 4.2　进程家谱示意图

这个组织虽然确定了每个进程的“宗族”关系，涵盖了系统中所有的进程，但是，要在这个组织中根据进程号 pid 找到一个进程却非易事。进程号的分配是相当随机的，在进程号中并不包含任何可以用来找到一个进程的路径信息，而给定一个进程号要求找到该进程的 task_struct 结构却又是常常要用到的一种操作。于是，就有了第二个组织，那就是一个以杂凑表为基础的进程队列的阵列。当给定一个 pid 要找到该进程时，先对 pid 施行杂凑计算，以计算的结果为下标在杂凑表中找到一个队列，再顺着该队列就可以较容易地找到特定的进程了。杂凑表 pidhash 是在 kernel/fork.c 中定义的：

==================== kernel/fork.c 35 35 ====================
```
35   struct task_struct *pidhash[PIDHASH_SZ];
```

杂凑表的大小 PIDHASH_SZ 则在 include/linux/sched.h 中定义：

==================== include/linux/sched.h 495 495 ====================
```
495  #define PIDHASH_SZ (4096 >> 2)
```

杂凑表的大小为 1024。由于每个指针的大小是 4 个字节，所以整个杂凑表（不包括各个队列）正好占一个页面。每个进程的 task_struct 数据结构都通过其 pidhash_next 和 pidhash_pprev 两个指针放入到杂凑表中的某个队列中，同一队列中所有进程的 pid 都具有相同的杂凑值。由于杂凑表的使用，要找到 pid 为某个给定值的进程就很迅速了。

当内核需要对每一个进程做点什么事情时，还需要将系统中所有的进程都组织成一个线性的队列，这样就可以通过一个简单的 for 循环或 while 循环遍历所有进程的 task_struct 结构。所以，第三个组织就是这么一个线性队列。系统中第一个建立的进程为 init_task，这个进程就是所有进程的总根，所以这个线性队列就是以 init_task 为起点（也可以把它看成是一个队列头），后继每创建一个进程就通过其 task_struct 结构中的 next_task 和 prev_task 两个指针链入这个线性队列中。

每个进程都必然同时身处这三个队列之中，直到进程消亡时才从这三个队列中摘除，所以这三个队列都是静态的。

在运行的过程中，一个进程还可以动态地链接进“可执行队列”接受系统的调度。实际上，这是最重要的队列，一个进程只有在可执行队列中才有可能受到调度而投入运行。与前几个队列不同的是，一个进程的 task_struct 是通过其 list_head 数据结构 run_list、而不是个别的指针，链接进可执行队列的。以前说过，这是用于双向链接的通用数据结构，具有一些与之配套的函数或宏操作，处理的效率比较高，也使代码得以简化。可执行队列的变化是非常频繁的，一个进程进入睡眠时就从队列中脱链，被唤醒时则又链入到该队列中，在调度的过程中也有可能会改变一个进程在此队列中的位置。详见本章“进程调度与进程切换”以及“系统调用 nanosleep()”中的有关叙述。

4.2　进程三部曲：创建、执行与消亡

就像世上万物都有产生、发展与消亡的过程一样，每个进程也有被创建、执行某段程序以及最后消亡的过程。在 Linux 系统中，第一个进程是系统固有的、与生俱来的或者说是由内核的设计者安排好了的。内核在引导并完成了基本的初始化以后，就有了系统的第一进程（实际上是内核线程）。除此之外，所有其它的进程和内核线程都由这个原始进程或其子孙进程所创建，都是这个原始进程的“后

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

代”。在 Linux 系统中，一个新的进程一定要由一个已经存在的进程“复制”出来，而不是“创造”出来（而所谓“创建”实际就是复制）。所以，Linux 系统（Unix 也一样）并不向用户（即进程）提供类似这样的系统调用：

```
int creat_proc(int (*fn)(void*), void *arg, unsigned long options);
```

可是在很多操作系统（包括一些 Unix 的变种）中都采用了“一揽子”的方法。它“创造”出一个进程，并使该进程从函数指针 fn 所指的地方开始执行。根据不同的情况和设计，参数 fn 也可以换成一个可执行程序的文件名。这里所谓“创造”，包括为进程分配所需的资源、包括属于最低限度的 task_struct 数据结构和系统空间堆栈，并初始化这些资源；还要设置其系统空间堆栈，使得这个新进程看起来就好像是一个本来就已经存在而正在睡眠的进程。当这个进程被调度运行的时候，其“返回地址”，也就是“恢复”运行时的下一条指令，则就在 fn 指的地方。这个“子进程”生下来时两手空空，却可以完全独立，并不与其父进程共享资源。

但是，Linux（以及 Unix）采用的方法却不同。

Linux 将进程的创建与目标程序的执行分成两步。第一步是从已经存在的“父进程”中像细胞分裂一样地复制出一个“子进程”。这里所谓像“细胞分裂一样”，只是打个比方，实际上，复制出来的子进程有自已的 task_struct 结构和系统空间堆栈，但与父进程共享其它所有的资源。例如，要是父进程打开了五个文件，那么子进程也有五个打开的文件，而且这些文件的当前读写指针也停在相同的地方。所以，这一步所做的是“复制”。Linux 为此提供了两个系统调用，一个是 fork()，另一个是 clone()。两者的区别在于 fork()是全部复制，父进程所有的资源全都通过数据结构的复制“遗传”给子进程。而 clone()则可以将资源有选择地复制给子进程，而没有复制的数据结构则通过指针的复制让子进程共享。在极端的情况下，一个进程可以 clone()出一个线程。所以，系统调用 fork()是无参数的，而 clone()则带有参数。读者也许已经意识到，fork()其实比 clone()更接近本来意义上的“克隆”。确实是这样，原因在于 fork()从 Unix 初期即已存在，那时候“克隆”这个词还不像现在这么流行，而既然业已存在，就不宜更改了。否则，也许应该互换一下名字。后来，又增设了一个系统调用 vfork()，也不带参数，但是除 task_struct 结构和系统空间堆栈以外的资源全都通过数据结构指针的复制“遗传”，所以 vfork()出来的是线程而不是进程。读者将会看到，vfork()主要是出于效率的考虑而设计并提供的。

第二步是目标程序的执行。一般来说，创建一个新的进程是因为有不同的目标程序要让新的程序去执行（但也不一定），所以，复制完成以后，子进程通常要与父进程分道扬镳，走自己的路。Linux 为此提供了一个系统调用 execve()，让一个进程执行以文件形式存在的一个可执行程序的映象。

读者也许要问：这两种方案到底哪一种好？应该说是各有利弊。但是更应该说，Linux 从 Unix 继承下来的这种分两步走，并且在第一步中采取复制方式的方案，利远大于弊。从效率的角度看，分两步走很有好处。所谓复制，只是进程的基本资源的复制，如 task_struct 数据结构、系统空间堆栈、页面表等等，对父进程的代码及全局变量则并不需要复制，而只是通过只读访问的形式实现共享，仅在需要写的时候才通过 copy_on_write 的手段为所涉及的页面建立一个新的副本。所以，总的来说复制的代价是很低的，但是通过复制而继承下来的资源则往往对子进程很有用。读者以后会看到，在计算机网络的实现中，以及在 client/server 系统中的 server 一方的实现中，fork()或 clone()常常是最自然、最有效、最适宜的手段。笔者有时候简直怀疑，到底是先有 fork()还是先有 client/server，因为 fork()似乎就是专门为此而设计的。更重要的好处是，这样有利于父、子进程间通过 pipe 来建立起一种简单有效

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的进程间通信管道，并且从而产生了操作系统的用户界面即 shell 的“管道”机制。这一点，对于 Unix 的发展和推广应用，对于 Unix 程序设计环境的形成，对于 Unix 程序设计风格的形成，都有着非常深远的影响。可以说，这是一项天才的发明，它在很大程度上改变了操作系统的发展方向。

当然，从另一角度，也就是从程序设计界面的角度来看，则“一揽子”的方案更为简洁。不过 fork()加 execve()的方案也并不复杂很多。进一步说，这也像练武或演戏一样有个固定的“招式”，一旦掌握了以后就不觉得复杂，也很少变化了。再说，如果有必要也可以通过程序库提供一个“一揽子”的库函数，将这两步包装在一起。

创建了子进程以后，父进程有三个选择。第一是继续走自己的路，与子进程分道扬镳。只是如果子进程先于父进程“去世”，则由内核给父进程发一个报丧的信号。第二是停下来，也就是进入睡眠状态，等待子进程完成其使命而最终去世，然后父进程再继续运行。Linux 为此提供了两个系统调用，wait4()和 wait3()。两个系统调用基本相同，wait4()等待某个特定的子进程去世，而 wait3()则等待任何一个子进程去世。第三个选择是“自行退出历史舞台”，结束自己的生命。Linux 为此设置了一个系统调用 exit()。这里的第三个选择其实不过是第一个选择的一种特例，所以从本质上说是两种选择：一种是父进程不受阻的（non_blocking）方式，也称为“异步”的方式；另一种是父进程受阻的（blocking）方式，或者称为“同步”的方式。

下面是一个用来演示进程的这种“生命周期”的简单程序：

```
1    #include <stdio.h>
2
3    int main()
4    {
5         int child;
6         char *args[] = { "/bin/echo", "Hello", "World!", NULL};
7
8         if (!(child = fork()))
9         {
10             /* child */
11             printf("pid %d: %d is my father\n", getpid(), getppid());
12             execve("/bin/echo", args, NULL);
13             printf("pid %d: I am back, something is wrong!\n", getpid());
14        }
15        else
16        {
17             int myself = getpid();
18             printf("pid %d: %d is my son\n", myself, child);
19             wait4(child, NULL, 0, NULL);
20             printf("pid %d: done\n", myself);
21        }
22        return 0;
23   }
```

这里，进入 main()的进程为父进程，它在第 8 行执行了系统调用 fork()创建一个子进程，也就是复制一个子进程。子进程复制出来以后，就像其父进程一样地接受内核的调度，而且具有相同的返回地址。所以，当父进程和子进程受调度继续运行而从内核空间返回时都返回到同一点上。以前的代码只

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

有一个进程执行，而从这一点开始却有两个进程在执行了。复制出来的子进程全面地继承了父进程的所有资源和特性，但还是有一些细微却重要的区别。首先，子进程有一个不同于父进程的进程号 pid，而且子进程的 task_struct 中有几个字段说明谁是它的父亲，就像人们的户口或档案中也有相应的栏目一样。其次，也许更为重要的是，二者从 fork()返回时所具有的返回值不一样。当子进程从 fork()“返回”时，其返回值为 0；而父进程从 fork()返回时的返回值却是子进程的 pid，这是不可能为 0 的。这样，第 8 行的 if 语句就可以根据这个特征把二者区分开来，使两个进程各自知道“我是谁”。然后，第 9～14 行属于子进程，而 16～21 行属于父进程，虽然两个进程具有相同的视野，都能“看到”对方所要执行的代码，但是 if 语句将它们各自的执行路线分开了。在这个程序中，我们选择了让父进程停下来等待，所以父进程执行 wait4()；而子进程则通过 execve()执行“ /bin/echo”。子进程在执行 echo 以后不会回到这里的第 13 行，而是“壮士一去不复返”。这是因为在/bin/echo 中必定有一个 exit()调用，使子进程结束它的生命。对 exit()的调用是每一个可执行程序映象必有的，虽然在我们这个程序中并没有调用它，而是以 return 语句从 main()返回，但是 gcc 在编译和连接时会自动加上，所以谁也逃不过这一关。

由于子进程与父进程一样接受内核调度，而每次系统调用都有可能引起调度，所以二者返回的先后次序是不定的，也不能根据返回的先后来确定谁是父进程谁是子进程。

还要指出，Linux 内核中确实有个貌似“一揽子”创建内核线程的函数（常常称为“原语”）kernel_thread()，供内核线程调用。但是，实际上这只是对 clone()的包装，它并不能像调用 execve()时那样执行一个可执行映象文件，而只是执行内核中的某一个函数。我们不妨看一下它的代码，这是在 arch/i386/kernel/process.c 给出的：

===================== arch/i386/kernel/process.c 439 463 =====================

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
	long retval, d0;

	__asm__ __volatile__(
		"movl %%esp,%%esi\n\t"
		"int $0x80\n\t"		/* Linux/i386 system call */
		"cmpl %%esp,%%esi\n\t"	/* child or parent? */
		"je 1f\n\t"		/* parent - jump */
		/* Load the argument into eax, and push it.  That way, it does
		 * not matter whether the called function is compiled with
		 * -mregparm or not.  */
		"movl %4,%%eax\n\t"
		"pushl %%eax\n\t"
		"call *%5\n\t"		/* call fn */
		"movl %3,%0\n\t"	/* exit */
		"int $0x80\n"
		"1:\t"
		:"=&a" (retval), "=&S" (d0)
		:"0" (__NR_clone), "i" (__NR_exit),
		 "r" (arg), "r" (fn),
		 "b" (flags | CLONE_VM)
		: "memory");
```

```
462       return retval;
463   }
```

这里 445 和 455 行的指令“int $0x80”就是系统调用。那么系统调用号是在哪里设置的呢？请看第 457 行的输出部，这里寄存器 EAX 与变量 retval 相结合作为%0，而在 458 行开始的输入部里又规定，%0 应事先赋值为__NR_clone。所以，在进入 444 行时寄存器 EAX 已经被设置成__NR_clone，即 clone()的系统调用号。从 clone()返回以后，这里采用了一种不同的方法区分父进程与子进程，就是将返回时的堆栈指针与保存在寄存器 ESI 中的父进程的堆栈指针进行比较。由于每一个内核线程都有自己的系统空间堆栈，子进程的堆栈指针必然与父进程不同。那么，为什么不采用像 fork()返回时所用的方法呢？这是因为 clone()所产生的子线程可以具有与父线程相同的 pid，如果 pid 为 0 的内核线程再 clone()一个子线程，则子线程的 pid 就也有可能是 0。所以，这里采用的比较堆栈指针的方法，是更为可靠的。当然，这个方法只有对内核线程才适用，因为普通的进程都在用户空间，根本就不知道其系统空间推栈到底在哪里。

前面讲过，内核线程不能像进程一样执行一个可执行映象文件，而只能执行内核中的一个函数。453 行的 call 指令就是对这个函数的调用。函数指针%5 是什么呢？从 457 行的输出部开始数一下，就可以知道%5 与变量 fn 相结合，而那正是 kernel_thread()的第一个参数。内核线程与进程在执行目标程序的方式上的这种不同，又引发出另一个重要的不同，那就是进程在调用 execve()之后不再返回，而是“客死他乡”，在所执行的程序中去世。可是内核线程只不过是调用一个目标函数，当然要从那个函数返回。所以，这里在 455 行又进行一次系统调用，而这次的系统调用号在%3 中，那是__NR_exit。

以后，我们将围绕着前面的那个程序来介绍系统调用 fork()、clone()、execve()、wait4()以及 exit()的实现，使读者对进程的创建、执行以及消亡有更深入的理解。

4.3 系统调用 fork()、vfork()与 clone()

前面已经简要地介绍过 fork()与 clone()二者的作用和区别。这里先来看一下二者在程序设计接口上的不同：

```
pid_t fork(void);
int clone(int (*fn)(void *arg), void *child_stack, int flags, void *arg);
```

系统调用__clone()的主要用途是创建一个线程，这个线程可以是内核线程，也可以是用户线程。创建用户空间线程时，可以给定子线程用户空间堆栈的位置，还可以指定子进程运行的起点。同时，也可以用__clone()创建进程，有选择地复制父进程的资源。而 fork()，则是全面地复制。还有一个系统调用 vfork()，其作用也是创建一个线程，但主要只是作为创建进程的中间步骤，目的在于提高创建时的效率，减少系统开销，其程序设计接口则与 fork 相同。

这几个系统调用的代码都在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 690 720 ====================

```
asmlinkage int sys_fork(struct pt_regs regs)
{
      return do_fork(SIGCHLD, regs.esp, &regs, 0);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}

asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;
    if (!newsp)
        newsp = regs.esp;
    return do_fork(clone_flags, newsp, &regs, 0);
}

/*
 * This is trivial, and on the face of it looks like it
 * could equally well be done in user mode.
 *
 * Not so, for quite unobvious reasons - register pressure.
 * In user mode vfork() cannot have a stack frame, and if
 * done by calling the "clone()" system call directly, you
 * do not have enough call-clobbered registers to hold all
 * the information you need.
 */
asmlinkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs, 0);
}
```

可见，三个系统调用的实现都是通过 do_fork()来完成的，不同的只是对 do_fork()的调用参数。关于这些参数所起的作用，读了 do_fork()的代码以后就会清楚。注意 sys_clone()中的 regs.ecx，就是调用__clone()时的参数 child_stack，读者如果还不清楚，可以回到第 3 章“系统调用”一节顺着代码再走一遍。调用__clone()时可以为子进程设置一个独立的用户空间堆栈（在同一个用户空间中），如果 child_stack 为 0，就表示使用父进程的用户空间堆栈。这三个系统调用的主体部分 do_fork()是在 kernel/fork.c 中定义的。这个函数比较大，让我们逐段往下看：

==================== kernel/fork.c 546 575 ====================

[sys_fork()>do_fork()]

```
/*
 *  Ok, this is the main fork-routine. It copies the system process
 * information (task[nr]) and sets up the necessary registers. It also
 * copies the data segment in its entirety.  The "stack_start" and
 * "stack_top" arguments are simply passed along to the platform
 * specific copy_thread() routine.  Most platforms ignore stack_top.
 * For an example that's using stack_top, see
 * arch/ia64/kernel/process.c.
 */
int do_fork(unsigned long clone_flags, unsigned long stack_start,
```

```
556          struct pt_regs *regs, unsigned long stack_size)
557  {
558      int retval = -ENOMEM;
559      struct task_struct *p;
560      DECLARE_MUTEX_LOCKED(sem);
561
562      if (clone_flags & CLONE_PID) {
563          /* This is only allowed from the boot up thread */
564          if (current->pid)
565              return -EPERM;
566      }
567
568      current->vfork_sem = &sem;
569
570      p = alloc_task_struct();
571      if (!p)
572          goto fork_out;
573
574      *p = *current;
575
```

第 560 行的宏操作 DECLARE_MUTEX_LOCKED()定义和创建了一个用于进程间互斥和同步的信号量，其定义和实现见第 6 章“进程间通信”。

参数 clone_flags 由两部分组成，其最低的字节为信号类型，用以规定子进程去世时应该向父进程发出的信号。我们已经看到，对于 fork()和 vfork()这个信号就是 SIGCHILD，而对__clone()则该位段可由调用者决定。第二部分是一些表示资源和特性的标志位，这些标志位是在 include/linux/sched.h 中定义的：

==================== include/linux/sched.h 30 44 ====================

```
/*
 * cloning flags:
 */
#define CSIGNAL         0x000000ff    /* signal mask to be sent at exit */
#define CLONE_VM    0x00000100    /* set if VM shared between processes */
#define CLONE_FS    0x00000200    /* set if fs info shared between processes */
#define CLONE_FILES     0x00000400    /* set if open files shared between processes */
#define CLONE_SIGHAND   0x00000800    /* set if signal handlers and blocked signals shared */
#define CLONE_PID   0x00001000    /* set if pid shared */
#define CLONE_PTRACE    0x00002000    /* set if we want to let tracing continue on the child too */
#define CLONE_VFORK     0x00004000    /* set if the parent wants the child to wake it up on mm_release */
#define CLONE_PARENT    0x00008000    /* set if we want to have the same parent as the cloner */
#define CLONE_THREAD    0x00010000    /* Same thread group? */

#define CLONE_SIGNAL    (CLONE_SIGHAND | CLONE_THREAD)
```

对于 fork()，这一部分为全 0，表现对有关的资源都要复制而不是通过指针共享。而对 vfork()，则为 CLONE_VFORK | CLONE_VM，表示父、子进程共用（用户）虚存区间，并且当子进程释放其虚存

PDF-XCHANGE Click to buy NOW! www.docu-track.com

区间时要唤醒父进程，至于__clone()，则这一部分完全由调用者设定而作为参数传递下来。其中标志位 CLONE_PID 有特殊的作用，当这个标志位为 1 时，父、子进程（线程）共用同一个进程号，也就是说，子进程虽然有其自己的 task_struct 数据结构，却使用父进程的 pid。但是，只有 0 号进程，也就是系统中的原始进程（实际上是线程），才允许这样来调用__clone()，所以 564 行对此加以检查。

接着，通过 alloc_task_struct()为子进程分配两个连续的物理页面，低端用作子进程的 task_struct 结构，高端则用作其系统空间堆栈。

注意 574 行的赋值为整个数据结构的赋值。这样，父进程的整个 task_struct 就被复制到了子进程的数据结构中。经编译以后，这样的赋值是用 memcpy()实现的，所以效率很高。

接着看下一段（kernel/fork.c）：

==================== kernel/fork.c 576 601 ====================

[sys_fork()>do_fork()]

```
	retval = -EAGAIN;
	if (atomic_read(&p->user->processes) >= p->rlim[RLIMIT_NPROC].rlim_cur)
		goto bad_fork_free;
	atomic_inc(&p->user->__count);
	atomic_inc(&p->user->processes);

	/*
	 * Counter increases are protected by
	 * the kernel lock so nr_threads can't
	 * increase under us (but it may decrease).
	 */
	if (nr_threads >= max_threads)
		goto bad_fork_cleanup_count;

	get_exec_domain(p->exec_domain);

	if (p->binfmt && p->binfmt->module)
		__MOD_INC_USE_COUNT(p->binfmt->module);

	p->did_exec = 0;
	p->swappable = 0;
	p->state = TASK_UNINTERRUPTIBLE;

	copy_flags(clone_flags, p);
	p->pid = get_pid(clone_flags);

```

在 task_struct 结构中有个指针 user，用来指向一个 user_struct 结构。一个用户常常有许多个进程，所以有关用户的一些信息并不专属于某一个进程。这样，属于同一用户的进程就可以通过指针 user 共享这些信息。显然，每个用户有且只有一个 user_struct 结构。结构中有个计数器__count，对属于该用户的进程数量计数。可想而知，内核线程并不属于某个用户，所以其 task_struct 中的 user 指引为 0。这个数据结构的定义在 include/linux/sched.h 中：

==================== include/linux/sched.h 256 267 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * Some day this will be a full-fledged user tracking system..
 */
struct user_struct {
	atomic_t __count;	/* reference count */
	atomic_t processes;	/* How many processes does this user have? */
	atomic_t files;		/* How many open files does this user have? */

	/* Hash table maintenance information */
	struct user_struct *next, **pprev;
	uid_t uid;
};
```

熟悉 Unix 内核的读者要注意，不要把 Unix 的进程控制结构中的 user 区与这里的 user_struct 结构相混淆，二者是截然不同的概念。在 kernel/user.c 中还定义了一个 user_struct 结构指针的数组 uidhash_table：

==================== kernel/user.c 19 20 ====================

```
#define UIDHASH_BITS		8
#define UIDHASH_SZ		(1 << UIDHASH_BITS)
```

==================== kernel/user.c 26 26 ====================

```
static struct user_struct *uidhash_table[UIDHASH_SZ];
```

这是一个杂凑（hash）表。对用户名施以杂凑运算，就可以计算出一个下标而找到该用户的 user_struct 结构。

各进程的 task_struct 结构中还有个数组 rlim，对该进程占用各种资源的数量作出限制，而 rlim[RLIMIT_NPROC]就规定了该进程所属的用户可以拥有的进程数量。所以，如果当前进程是一个用户进程，并且该用户拥有的进程数量已经达到了规定的限制值，就再不允许它 fork()了。那么，对于不属于任何用户的内核线程怎么办呢？587 行中的两个计数器就是为进程的总量而设的。

一个进程除了属于某一个用户之外，还属于某个“执行域”。总的来说，Linux 是 Unix 的一个变种，并且符合 POSIX 的规定。但是，有很多版本的操作系统同样是 Unix 变种，同样符合 POSIX 规定，互相之间在实现细节上却仍然有明显的不同。例如，AT&T 的 Sys V 和 BSD 4.2 就有相当的不同，而 Sun 的 Solaris 又有区别，这就形成了不同的执行域。如果一个进程所执行的程序是为 Solaris 开发的，那么这个进程就属于 Solaris 执行域 PER_SOLARIS。当然，在 Linux 上运行的绝大多数程序都属于 Linux 执行域。在 task_struct 结构中有一个指针 exec_domain，可以指向一个 exec_domain 数据结构。那是在 include/linux/personality.h 中定义的：

==================== include/linux/personality.h 38 51 ====================

```
/* Description of an execution domain - personality range supported,
 * lcall7 syscall handler, start up / shut down functions etc.
 * N.B. The name and lcall7 handler must be where they are since the
 * offset of the handler is hard coded in kernel/sys_call.S.
 */
struct exec_domain {
	const char *name;
	lcall7_func handler;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        unsigned char pers_low, pers_high;
        unsigned long * signal_map;
        unsigned long * signal_invmap;
        struct module * module;
        struct exec_domain *next;
    };
```

函数指针 handler，用于通过调用门实现系统调用，我们并不关心。字节 pers_low 为某种域的代码，有 PER_LINUX、PER_SVR4、PER_BSD 和 PER_SOLARIS 等等。

我们在这里主要关心的结构成分是 module，这是指向某个 module 数据结构的指针。读者在有关文件系统和设备驱动的章节中将会看到，在 Linux 系统中设备驱动程可以设计并实现成“动态安装模块”module，使其在运行时动态地安装和拆除。这些“动态安装模块”与运行中的进程的执行域有密切的关系。例如，一个属于 Solaris 执行域的进程就很可能要用到专门为 Solaris 设置的一些模块，只要还有一个这样的进程在运行；这些为 Solaris 所需的模块就不能拆除。所以，在描述每个已安装模块的数据结构中都有一个计数器，表明有几个进程需要使用这个模块。因此，do_fork()中通过 590 行的 get_exec_domain()递增具体模块的数据结构中的计数器（定义在 include/linux/personality.h 中）。

==================== include/linux/personality.h 59 60 ====================

```
    #define get_exec_domain(it) \
        if (it && it->module) __MOD_INC_USE_COUNT(it->module);
```

同样的道理，每个进程所执行的程序属于某种可执行映象格式，如 a.out 格式、elf 格式、甚至 java 虚拟机格式。对这些不同格式的支持通常是通过动态安装的驱动模块来实现的。所以 task_struct 结构中还有一个指向 Linux_binfmt 数据结构的指针 binfmt，而 do_fork()中 593 行的 __MOD_INC_USE_COUNT()就是对有关模块的使用计数器进行操作。

为什么要在 597 行把状态设成 TASK_UNINTERRUPTIBLE 呢？这是因为在 get_pid()中产生一个新 pid 的操作必须是独占的，当前进程可能会因为一时进不了临界区而只好暂时进入睡眠状态等待，所以才事先把状态设成 UNINTERRUPTIBLE。函数 copy_flags()将参数 clone_flags 中的标志位略加补充和变换，然后写入 p->flags。这个函数的代码也在 kernel/fork.c 中。读者可以自己阅读。

至于 600 行的 get_pid()，则根据 clone_flags 中标志位 CLONE_PID 的值，或返回父进程（当前进程）的 pid，或返回一个新的 pid 放在子进程的 task_struct 中。函数 get_pid()的代码也在 kernel/fork.c 中：

==================== kernel/fork.c 82 123 ====================

[sys_fork()>do_fork()>get_pid()]

```
    static int get_pid(unsigned long flags)
    {
        static int next_safe = PID_MAX;
        struct task_struct *p;

        if (flags & CLONE_PID)
            return current->pid;

        spin_lock(&lastpid_lock);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	if((++last_pid) & 0xffff8000) {
		last_pid = 300;		/* Skip daemons etc. */
		goto inside;
	}
	if(last_pid >= next_safe) {
inside:
		next_safe = PID_MAX;
		read_lock(&tasklist_lock);
	repeat:
		for_each_task(p) {
			if(p->pid == last_pid	||
			   p->pgrp == last_pid	||
			   p->session == last_pid) {
				if(++last_pid >= next_safe) {
					if(last_pid & 0xffff8000)
						last_pid = 300;
					next_safe = PID_MAX;
				}
				goto repeat;
			}
			if(p->pid > last_pid && next_safe > p->pid)
				next_safe = p->pid;
			if(p->pgrp > last_pid && next_safe > p->pgrp)
				next_safe = p->pgrp;
			if(p->session > last_pid && next_safe > p->session)
				next_safe = p->session;
		}
		read_unlock(&tasklist_lock);
	}
	spin_unlock(&lastpid_lock);

	return last_pid;
}
```

这里的常数 PID_MAX 定义为 0x8000。可见，进程号的最大值是 0x7fff，即 32767。进程号 0～299 是为系统进程（包括内核线程）保留的，主要用于各种“保护神”进程。以上这段代码的逻辑并不复杂，我们就不多加解释了。

回到 do_fork()中再往下看（kernel/fork.c）：

==================== kernel/fork.c 602 640 ====================

[sys_fork()>do_fork()]

```
	p->run_list.next = NULL;
	p->run_list.prev = NULL;

	if ((clone_flags & CLONE_VFORK) || !(clone_flags & CLONE_PARENT)) {
		p->p_opptr = current;
		if (!(p->ptrace & PT_PTRACED))
			p->p_pptr = current;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
609         }
610         p->p_cptr = NULL;
611         init_waitqueue_head(&p->wait_chldexit);
612         p->vfork_sem = NULL;
613         spin_lock_init(&p->alloc_lock);
614
615         p->sigpending = 0;
616         init_sigpending(&p->pending);
617
618         p->it_real_value = p->it_virt_value = p->it_prof_value = 0;
619         p->it_real_incr = p->it_virt_incr = p->it_prof_incr = 0;
620         init_timer(&p->real_timer);
621         p->real_timer.data = (unsigned long) p;
622
623         p->leader = 0;          /* session leadership doesn't inherit */
624         p->tty_old_pgrp = 0;
625         p->times.tms_utime = p->times.tms_stime = 0;
626         p->times.tms_cutime = p->times.tms_cstime = 0;
627     #ifdef CONFIG_SMP
628         {
629             int i;
630             p->has_cpu = 0;
631             p->processor = current->processor;
632             /* ?? should we just memset this ?? */
633             for(i = 0; i < smp_num_cpus; i++)
634                 p->per_cpu_utime[i] = p->per_cpu_stime[i] = 0;
635             spin_lock_init(&p->sigmask_lock);
636         }
637     #endif
638         p->lock_depth = -1;         /* -1 = no lock */
639         p->start_time = jiffies;
640
```

我们在前一节中提到过 wait4()和 wait3()，一个进程可以停下来等待其子进程完成使命。为此，在 task_struct 中设置了一个队列头部 wait_chldexit，前面在复制 task_struct 结构时把这也照抄了过来，而子进程此时尚未“出生”，当然谈不上子进程的等待队列，所以要在 611 行中加以初始化。

类似地，对各种信息量也要加以初始化。这里 615 和 616 行是对子进程的待处理信号队列以及有关结构成分的初始化。对这些与信号有关的结构成分我们将在“进程间通信”的信号一节中详细介绍。接下来是对 task_struct 结构中各种计时变量的初始化，我们将在“进程调度”一书中介绍这些变量。在这里我们不关心对多处理器 SMP 结构的特殊考虑，所以也跳过 627～637 行。

最后，task_struct 结构中的 start_time 表示进程创建的时间，而全局变量 jiffies 的数值就是以时钟中断周期为单位的从系统初始化开始至此时的时间。

至此，对 task_struct 数据结构的复制与初始化就基本完成了。下由就轮到其它的资源了：

==================== kernel/fork.c 641 655 ====================

[sys_fork()>do_fork()]

```
641         retval = -ENOMEM;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	/* copy all the process information */
	if (copy_files(clone_flags, p))
		goto bad_fork_cleanup;
	if (copy_fs(clone_flags, p))
		goto bad_fork_cleanup_files;
	if (copy_sighand(clone_flags, p))
		goto bad_fork_cleanup_fs;
	if (copy_mm(clone_flags, p))
		goto bad_fork_cleanup_sighand;
	retval = copy_thread(0, clone_flags, stack_start, stack_size, p, regs);
	if (retval)
		goto bad_fork_cleanup_sighand;
	p->semundo = NULL;

```

函数 copy_files()有条件地复制已打开文件的控制结构，这种复制只有在 clone_flags 中 CLONE_FILES 标志位为 0 时才真正进行，否则就只是共享父进程的已打开文件。当一个进程有已打开文件时，task_struct 结构中的指针 files 指向一个 files_struct 数据结构，否则为 0。所有与终端设备 tty 相联系的用户进程的头三个文件，即 stdin、stdout、及 stderr，都是预先打开的，所以指针一般不会是 0。数据结构 files_struct 是在 include/linux/sched.h 中定义的（详见“文件系统”一章），copy_files()的代码则还是在 kernel/fork.c 中：

==================== kernel/fork.c 408 513 ====================
[sys_fork()>do_fork()>copy_files]

```
static int copy_files(unsigned long clone_flags, struct task_struct * tsk)
{
	struct files_struct *oldf, *newf;
	struct file **old_fds, **new_fds;
	int open_files, nfds, size, i, error = 0;

	/*
	 * A background process may not have any files ...
	 */
	oldf = current->files;
	if (!oldf)
		goto out;

	if (clone_flags & CLONE_FILES) {
		atomic_inc(&oldf->count);
		goto out;
	}

	tsk->files = NULL;
	error = -ENOMEM;
	newf = kmem_cache_alloc(files_cachep, SLAB_KERNEL);
	if (!newf)
		goto out;

```

```
	atomic_set(&newf->count, 1);

	newf->file_lock	    = RW_LOCK_UNLOCKED;
	newf->next_fd	    = 0;
	newf->max_fds	    = NR_OPEN_DEFAULT;
	newf->max_fdset	    = __FD_SETSIZE;
	newf->close_on_exec = &newf->close_on_exec_init;
	newf->open_fds	    = &newf->open_fds_init;
	newf->fd	    = &newf->fd_array[0];

	/* We don't yet have the oldf readlock, but even if the old
	   fdset gets grown now, we'll only copy up to "size" fds */
	size = oldf->max_fdset;
	if (size > __FD_SETSIZE) {
		newf->max_fdset = 0;
		write_lock(&newf->file_lock);
		error = expand_fdset(newf, size);
		write_unlock(&newf->file_lock);
		if (error)
			goto out_release;
	}
	read_lock(&oldf->file_lock);

	open_files = count_open_files(oldf, size);

	/*
	 * Check whether we need to allocate a larger fd array.
	 * Note: we're not a clone task, so the open count won't
	 * change.
	 */
	nfds = NR_OPEN_DEFAULT;
	if (open_files > nfds) {
		read_unlock(&oldf->file_lock);
		newf->max_fds = 0;
		write_lock(&newf->file_lock);
		error = expand_fd_array(newf, open_files);
		write_unlock(&newf->file_lock);
		if (error)
			goto out_release;
		nfds = newf->max_fds;
		read_lock(&oldf->file_lock);
	}

	old_fds = oldf->fd;
	new_fds = newf->fd;

	memcpy(newf->open_fds->fds_bits, oldf->open_fds->fds_bits, open_files/8);
	memcpy(newf->close_on_exec->fds_bits, oldf->close_on_exec->fds_bits, open_files/8);
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
	for (i = open_files; i != 0; i--) {
		struct file *f = *old_fds++;
		if (f)
			get_file(f);
		*new_fds++ = f;
	}
	read_unlock(&oldf->file_lock);

	/* compute the remainder to be cleared */
	size = (newf->max_fds - open_files) * sizeof(struct file *);

	/* This is long word aligned thus could use a optimized version */
	memset(new_fds, 0, size);

	if (newf->max_fdset > open_files) {
		int left = (newf->max_fdset-open_files)/8;
		int start = open_files / (8 * sizeof(unsigned long));

		memset(&newf->open_fds->fds_bits[start], 0, left);
		memset(&newf->close_on_exec->fds_bits[start], 0, left);
	}

	tsk->files = newf;
	error = 0;
out:
	return error;

out_release:
	free_fdset (newf->close_on_exec, newf->max_fdset);
	free_fdset (newf->open_fds, newf->max_fdset);
	kmem_cache_free(files_cachep, newf);
	goto out;
}
```

读者可以在学习了“文件系统”一章以后再回过头来仔细阅读这段代码，我们在这里先作一些解释。

先看复制的方向。因为是当前进程在创建子进程，是从当前进程复制到子进程，所以把当前进程的 task_struct 结构中的 files_struct 结构指针作为 oldf。

再看复制的条件。如果参数 clone_flags 中的 CLONE_FILES 标志位为 1，就只是通过 atomic_inc() 递增当前进程的 files_struct 结构中的共享计数，表示这个数据结构现在多了一个“用户”，就返回了。由于在此之前已通过数据结构赋值将当前进程的整个 task_struct 结构都复制给了子进程，结构中的指针 files 自然也复制到了子进程的 task_struct 结构中，使子进程通过这个指针共享当前进程的 files_struct 数据结构。否则，如果 CLONE_FILES 标志位为 0，那就要复制了。首先通过 kmem_cache_alloc()为子进程分配一个 files_struct 数据结构作为 newf，然后从 oldf 把内容复制到 newf。在 files_struct 数据结构中有三个主要的“部件”。其一是个位图，名为 close_on_exec_init；其二也是位图，名为 open_fds_init；其三则是 file 结构数组 fd_array[]。这三个部件都是固定大小的，如果打开的文件数量超过其容量，就

PDF-XCHANGE Click to buy NOW! www.docu-track.com

得通过 expand_fdset()和 expand_fd_array()在 files_struct 数据结构以外另行分配空间作为替换。不管是采用 files_struct 数据结构内部的这三个部件或是采用外部的替换空间，指针 close_on_exec、open_fds 和 fd 总是分别指向这三组信息。所以，如何复制取决于已打开文件的数量。

显而易见，共享比复制要简单得多。那么这二者在效果上到底有什么区别呢？如用共享就可以达到目的，为什么还要不辞辛劳地复制呢？区别在于子进程（以及父进程本身）是否能“独立自主”。当复制完成之初，子进程有了一份副本，它的内容与父进程的“正本”在内容上基本是相同的，在这一点上似乎与共享没有什么区别。可是，随后区别就来了。在共享的情况下，两个进程是互相牵制的。如果子进程对某个已打开文件调用了一次 lseek()，则父进程对这个文件的读写位置也随着改变了，因为两个进程共享着对文件的同一个读写上下文。而在复制的情况下就不一样了，由于子进程有自己的副本，就有了对同一文件的另一个读写上下文，以后就可以各走各的路，互不干扰了。

除 files_struct 数据结构外，还有个 fs_struct 数据结构也是与文件系统有关的，也要通过共享或复制遗传给子进程。类似地，copy_fs()也是只有在 clone_flags 中 CLONE_FS 标志位为 0 时才加以复制。task_struct 结构中的指针指向一个 fs_struct 数据结构，结构中记录的是进程的根目录 root、当前工作目录 pwd、一个用于文件操作权限管理的 umask，还有一个计数器，其定义在 include/linux/fs_struct.h 中（详见“文件系统”一章）。函数 copy_fs()连同几个有关低层函数的代码也在 kernel/fork.c 中。我们把这些代码留给读者：

==================== kernel/fork.c 383 393 ====================

[sys_fork()>do_fork()>copy_fs()]

```
static inline int copy_fs(unsigned long clone_flags, struct task_struct * tsk)
{
	if (clone_flags & CLONE_FS) {
		atomic_inc(&current->fs->count);
		return 0;
	}
	tsk->fs = __copy_fs_struct(current->fs);
	if (!tsk->fs)
		return -1;
	return 0;
}
```

==================== kernel/fork.c 378 381 ====================

[sys_fork()>do_fork()>copy_fs()>copy_fs_struct()]

```
struct fs_struct *copy_fs_struct(struct fs_struct *old)
{
	return __copy_fs_struct(old);
}
```

==================== kernel/fork.c 353 376 ====================

[sys_fork()>do_fork()>copy_fs()>copy_fs_struct()>__cooy_fs_struct()]

```
static inline struct fs_struct *__copy_fs_struct(struct fs_struct *old)
{
	struct fs_struct *fs = kmem_cache_alloc(fs_cachep, GFP_KERNEL);
	/* We don't need to lock fs - think why ;-) */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (fs) {
         atomic_set(&fs->count, 1);
         fs->lock = RW_LOCK_UNLOCKED;
         fs->umask = old->umask;
         read_lock(&old->lock);
         fs->rootmnt = mntget(old->rootmnt);
         fs->root = dget(old->root);
         fs->pwdmnt = mntget(old->pwdmnt);
         fs->pwd = dget(old->pwd);
         if (old->altroot) {
              fs->altrootmnt = mntget(old->altrootmnt);
              fs->altroot = dget(old->altroot);
         } else {
              fs->altrootmnt = NULL;
              fs->altroot = NULL;
         }
         read_unlock(&old->lock);
    }
    return fs;
}
```

代码中的 mntget()和 dget()都是用来递增相应数据结构中共享计数的，因为这些数据结构现在多了一个用户。注意，在这里要复制的是 fs_struct 数据结构，而并不复制更深层的数据结构。复制了 fs_struct 数据结构，就在这一层上有了自主性，至于对更深层的数据结构则还是共享，所以要递增它们的共享计数。

接着是关于对信号的处理方式。是否复制父进程对信号的处理是由标志位 CLONE_SIGHAND 控制的。信号基本上是一种进程间通信手段，信号之于一个进程就好像中断之于一个处理器。进程可以为各种信号设置用于该信号的处理程序，就好像系统可以为各个中断源设置相应的中断服务程序一样。如果一个进程设置了信号处理程序，其 task_struct 结构中的指针 sig 就指向一个 signal_struct 数据结构。这种结构是在 include/linux/sched.h 中定义的：

==================== include/linux/sched.h 243 247 ====================

```
struct signal_struct {
    atomic_t       count;
    struct k_sigaction action[_NSIG];
    spinlock_t          siglock;
};
```

其中的数组 action[]确定了一个进程对各种信号（以信号的数值为下标）的反应和处理，子进程可以通过复制或共享把它从父进程继承下来。函数 copy_sighand()的代码如下（kernel/fork.c）：

==================== kernel/fork.c 515 531 ====================

[sys_fork()>do_fork()>copy_sighand()]

```
static inline int copy_sighand(unsigned long clone_flags, struct task_struct * tsk)
{
    struct signal_struct *sig;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (clone_flags & CLONE_SIGHAND) {
        atomic_inc(&current->sig->count);
        return 0;
    }
    sig = kmem_cache_alloc(sigact_cachep, GFP_KERNEL);
    tsk->sig = sig;
    if (!sig)
        return -1;
    spin_lock_init(&sig->siglock);
    atomic_set(&sig->count, 1);
    memcpy(tsk->sig->action, current->sig->action, sizeof(tsk->sig->action));
    return 0;
}
```

像 copy_files()和 copy_fs()一样，copy_sighand()也是只有在 CLONE_SIGHAND 为 0 时才真正进行；否则就共享父进程的 sig 指针，并将父进程的 signal_struct 中的共享计数加 1。

然后是用户空间的继承。进程的 task_struct 结构中有个指针 mm，读者已经相当熟悉了，它指向一个代表着进程的用户空间的 mm_struct 数据结构。由于内核线程并不拥有用户空间，所以在内核线程的 task_struct 结构中该指针为 0。有关 mm_struct 及其下属的 vm_area_struct 等数据结构已经在第 2 章中介绍过，这里不再重复。函数 copy_mm()的代码还是在 kernel/fork.c 中：

==================== kernel/fork.c 279 351 ====================

[sys_fork()>do_fork()>copy_mm()]

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
    struct mm_struct * mm, *oldmm;
    int retval;

    tsk->min_flt = tsk->maj_flt = 0;
    tsk->cmin_flt = tsk->cmaj_flt = 0;
    tsk->nswap = tsk->cnswap = 0;

    tsk->mm = NULL;
    tsk->active_mm = NULL;

    /*
     * Are we cloning a kernel thread?
     *
     * We need to steal a active VM for that..
     */
    oldmm = current->mm;
    if (!oldmm)
        return 0;

    if (clone_flags & CLONE_VM) {
        atomic_inc(&oldmm->mm_users);
        mm = oldmm;
        goto good_mm;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    }

    retval = -ENOMEM;
    mm = allocate_mm();
    if (!mm)
        goto fail_nomem;

    /* Copy the current MM stuff.. */
    memcpy(mm, oldmm, sizeof(*mm));
    if (!mm_init(mm))
        goto fail_nomem;

    down(&oldmm->mmap_sem);
    retval = dup_mmap(mm);
    up(&oldmm->mmap_sem);

    /*
     * Add it to the mmlist after the parent.
     *
     * Doing it this way means that we can order
     * the list, and fork() won't mess up the
     * ordering significantly.
     */
    spin_lock(&mmlist_lock);
    list_add(&mm->mmlist, &oldmm->mmlist);
    spin_unlock(&mmlist_lock);

    if (retval)
        goto free_pt;

    /*
     * child gets a private LDT (if there was an LDT in the parent)
     */
    copy_segments(tsk, mm);

    if (init_new_context(tsk,mm))
        goto free_pt;

good_mm:
    tsk->mm = mm;
    tsk->active_mm = mm;
    return 0;

free_pt:
    mmput(mm);
fail_nomem:
    return retval;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

显然，对 mm_struct 的复制也是只在 clone_flags 中 CLONE_VM 标志为 0 时才真正进行，否则就只是通过已经复制的指针共享父进程的用户空间。对 mm_struct 的复制就不只是局限于这个数据结构本身了，也包括了对更深层数据结构的复制。其中最重要的是 vm_area_struct 数据结构和页面映射表，这是由 dup_mmap()复制的。函数 dup_mmap()的代码也在 kernel/fork.c 中。读者在认真读过本书第 2 章以后，阅读这段程序时应该不会感到困难，同时也是一次很好的练习。

==================== kernel/fork.c 125 193 ====================

[sys_fork()>do_fork()>copy_mm()>dup_mmap()]

```
static inline int dup_mmap(struct mm_struct * mm)
{
	struct vm_area_struct * mpnt, *tmp, **pprev;
	int retval;

	flush_cache_mm(current->mm);
	mm->locked_vm = 0;
	mm->mmap = NULL;
	mm->mmap_avl = NULL;
	mm->mmap_cache = NULL;
	mm->map_count = 0;
	mm->cpu_vm_mask = 0;
	mm->swap_cnt = 0;
	mm->swap_address = 0;
	pprev = &mm->mmap;
	for (mpnt = current->mm->mmap ; mpnt ; mpnt = mpnt->vm_next) {
		struct file *file;

		retval = -ENOMEM;
		if(mpnt->vm_flags & VM_DONTCOPY)
			continue;
		tmp = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
		if (!tmp)
			goto fail_nomem;
		*tmp = *mpnt;
		tmp->vm_flags &= ~VM_LOCKED;
		tmp->vm_mm = mm;
		mm->map_count++;
		tmp->vm_next = NULL;
		file = tmp->vm_file;
		if (file) {
			struct inode *inode = file->f_dentry->d_inode;
			get_file(file);
			if (tmp->vm_flags & VM_DENYWRITE)
				atomic_dec(&inode->i_writecount);

			/* insert tmp into the share list, just after mpnt */
			spin_lock(&inode->i_mapping->i_shared_lock);
			if((tmp->vm_next_share = mpnt->vm_next_share) != NULL)
				mpnt->vm_next_share->vm_pprev_share =
```

```
165                         &tmp->vm_next_share;
166                 mpnt->vm_next_share = tmp;
167                 tmp->vm_pprev_share = &mpnt->vm_next_share;
168                 spin_unlock(&inode->i_mapping->i_shared_lock);
169             }
170 
171             /* Copy the pages, but defer checking for errors */
172             retval = copy_page_range(mm, current->mm, tmp);
173             if (!retval && tmp->vm_ops && tmp->vm_ops->open)
174                 tmp->vm_ops->open(tmp);
175 
176             /*
177              * Link in the new vma even if an error occurred,
178              * so that exit_mmap() can clean up the mess.
179              */
180             *pprev = tmp;
181             pprev = &tmp->vm_next;
182 
183             if (retval)
184                 goto fail_nomem;
185         }
186         retval = 0;
187         if (mm->map_count >= AVL_MIN_MAP_COUNT)
188             build_mmap_avl(mm);
189 
190     fail_nomem:
191         flush_tlb_mm(current->mm);
192         return retval;
193     }
```

这里通过 140～185 行的 for 循环对同一用户空间中的各个区间进行复制。对于通过 mmap()映射到某个文件的区间，155～169 行是一些特殊的附加处理。172 行的 copy_page_range()是关键所在，这个函数逐层处理页面目录项和页面表项，其代码在 mm/memory.c 中：

==================== mm/memory.c 144 257 ====================

[sys_fork()>do_fork()>copy_mm()>dup_mmap()>copy_page_range()]

```
/*
 * copy one vm_area from one task to the other. Assumes the page tables
 * already present in the new task to be cleared in the whole range
 * covered by this vma.
 *
 * 08Jan98 Merged into one routine from several inline routines to reduce
 *         variable count and make things faster. -jj
 */
int copy_page_range(struct mm_struct *dst, struct mm_struct *src,
            struct vm_area_struct *vma)
{
    pgd_t * src_pgd, * dst_pgd;
    unsigned long address = vma->vm_start;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
	unsigned long end = vma->vm_end;
	unsigned long cow = (vma->vm_flags & (VM_SHARED | VM_MAYWRITE)) == VM_MAYWRITE;

	src_pgd = pgd_offset(src, address)-1;
	dst_pgd = pgd_offset(dst, address)-1;

	for (;;) {
		pmd_t * src_pmd, * dst_pmd;

		src_pgd++; dst_pgd++;

		/* copy_pmd_range */

		if (pgd_none(*src_pgd))
			goto skip_copy_pmd_range;
		if (pgd_bad(*src_pgd)) {
			pgd_ERROR(*src_pgd);
			pgd_clear(src_pgd);
skip_copy_pmd_range:	address = (address + PGDIR_SIZE) & PGDIR_MASK;
			if (!address || (address >= end))
				goto out;
			continue;
		}
		if (pgd_none(*dst_pgd)) {
			if (!pmd_alloc(dst_pgd, 0))
				goto nomem;
		}

		src_pmd = pmd_offset(src_pgd, address);
		dst_pmd = pmd_offset(dst_pgd, address);

		do {
			pte_t * src_pte, * dst_pte;

			/* copy_pte_range */

			if (pmd_none(*src_pmd))
				goto skip_copy_pte_range;
			if (pmd_bad(*src_pmd)) {
				pmd_ERROR(*src_pmd);
				pmd_clear(src_pmd);
skip_copy_pte_range:		address = (address + PMD_SIZE) & PMD_MASK;
				if (address >= end)
					goto out;
				goto cont_copy_pmd_range;
			}
			if (pmd_none(*dst_pmd)) {
				if (!pte_alloc(dst_pmd, 0))
					goto nomem;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                }

                src_pte = pte_offset(src_pmd, address);
                dst_pte = pte_offset(dst_pmd, address);

                do {
                        pte_t pte = *src_pte;
                        struct page *ptepage;

                        /* copy_one_pte */

                        if (pte_none(pte))
                                goto cont_copy_pte_range_noset;
                        if (!pte_present(pte)) {
                                swap_duplicate(pte_to_swp_entry(pte));
                                goto cont_copy_pte_range;
                        }
                        ptepage = pte_page(pte);
                        if ((!VALID_PAGE(ptepage)) ||
                            PageReserved(ptepage))
                                goto cont_copy_pte_range;

                        /* If it's a COW mapping, write protect it both in the parent and the child */
                        if (cow) {
                                ptep_set_wrprotect(src_pte);
                                pte = *src_pte;
                        }

                        /* If it's a shared mapping, mark it clean in the child */
                        if (vma->vm_flags & VM_SHARED)
                                pte = pte_mkclean(pte);
                        pte = pte_mkold(pte);
                        get_page(ptepage);

cont_copy_pte_range:            set_pte(dst_pte, pte);
cont_copy_pte_range_noset:      address += PAGE_SIZE;
                        if (address >= end)
                                goto out;
                        src_pte++;
                        dst_pte++;
                } while ((unsigned long)src_pte & PTE_TABLE_MASK);

cont_copy_pmd_range:    src_pmd++;
                dst_pmd++;
        } while ((unsigned long)src_pmd & PMD_TABLE_MASK);
    }
out:
    return 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
255 nomem:
256         return -ENOMEM;
257 }
```

代码中 163 行的 for 循环是对页面目录项的循环，188 行的 do 循环是对中间目录项的循环，211 行的 do 循环则是对页面表项的循环。我们把注意力集中在 211～246 行对页面表项的 do-while 循环。

循环中检查父进程一个页面表中的每个表项，根据表项的内容决定具体的操作。而表项的内容，则无非是下面这么一些可能：

(1) 表项的内容为全 0，所以 pte_none()返回 1。说明该页面的映射尚未建立，或者说是个“空洞”，因此不需要做任何事。

(2) 表项的最低位，即_PAGE_PRESENT 标志位为 0，所以 pte_present()返回 1。说明映射已建立，但是该页面目前不在内存中，已经被调出到交换设备上。此时表项的内容指明“盘上页面”的地点，而现在该盘上页面多了一个“用户”，所以要通过 swap_duplicate()递增它的共享计数。然后，就转到 cont_copy_pte_range 将此表项复制到子进程的页面表中。

(3) 映射已建立，但是物理页面不是一个有效的内存页面，所以 VALID_PAGE()返回 0。读者可以回顾一下，我们以前讲过有些物理页面在外设接口卡上，相应的地址称为“总线地址”，而并不是内存页面。这样的页面、以及虽是内存页面但由内核保留的页面，是不属于页面换入／换出机制管辖的，实际上也不消耗动态分配的内存页面，所以也转到 cont_copy_pte_range 将此表项复制到子进程的页面表中。

(4) 需要从父进程复制的可写页面。本来，此时应该分配一个空闲的内存页面，再从父进程的页面把内容复制过来，并为之建立映射。显然，这个操作的代价是不小的。然而，对这么辛辛苦苦复制下来的页面，子进程是否一定会用呢？特别是会有写访问呢？如果只是读访问，则只要父进程从此不再写这个页面，就完全可以通过复制指针来共享这个页面，那不知要省事多少了。所以，Linux 内核采用了一种称为“copy on write”的技术，先通过复制页面表项暂时共享这个页面，到子进程（或父进程）真的要写这个页面时再次分配页面和复制。代码中的局部变量 cow 是在前面 158 行定义的，变量名 cow 是“copy on write”的缩写。只要一个虚存区间的性质是可写（VM_MAYWRITE 为 1）而又不是共享（VM_SHARED 为 0），就属于 copy_on_write 区间。实际上，对于绝大多数的可写虚存区间，cow 都是 1。在通过复制页面表项暂时共享一个页面表项时要做两件重要的事情，首先要在 230 和 231 行将父进程的页面表项改成写保护，然后在 236 行把已经改成写保护的表项设置到子进程的页面表中。这样一来，相应的页面在两个进程中都变成“只读”了，当不管是父进程或是子进程企图写入该页面时，都会引起一次页面异常。而页面异常处理程序对此的反应则是另行分配一个物理页面，并把内容真正地“复制”到新的物理页面中，，让父、子进程各自拥有自己的物理页面，然后将两个页面表中相应的表项改成可写。所以，Linux 内核之所以可以很迅速地“复制”一个进程，完全依赖于“copy on wrtre”（否则，在 fork 一个进程时就得要复制每一个物理页面了）。可是，copy_on_write 只有在父、子进程各自拥有自己的页面表时才能实现。当 CLONE_VM 标志位为 1，因而父、子进程通过指针共享用户空间时，copy_on_write 就用不上了。此时，父、子进程是在真正的意义上共享用户空间，父进程写入其用户空间的内容同时也“写入”子进程的用户空间。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

(5) 父进程的只读页面。这种页面本来就不需要复制。因而可以复制页面表项共享物理页面。

可见，名为 copy_page_range()，实际上却连一个页面也没有真正地“复制”，这就是为什么 Linux 内核能够很迅速地 fork()或 clone()一个进程的秘密。

回到 copy_mm()的代码中。函数 copy_segments()处理的是进程可能具有的局部段描述表 LDT。我们在第 2 章中讲过，只有在 VM86 模式中运行的进程才会有 LDT。虽然我们并不关心 VM86 模式，但是有兴趣的读者也不妨自己看看它是怎样复制的。copy_segments()的代码在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 499 521 ====================
[sys_fork()>do_fork()>copy_mm()>copy_segments()]

```
/*
 * we do not have to muck with descriptors here, that is
 * done in switch_mm() as needed.
 */
void copy_segments(struct task_struct *p, struct mm_struct *new_mm)
{
     struct mm_struct * old_mm;
     void *old_ldt, *ldt;

     ldt = NULL;
     old_mm = current->mm;
     if (old_mm && (old_ldt = old_mm->context.segments) != NULL) {
          /*
           * Completely new LDT, we initialize it from the parent:
           */
          ldt = vmalloc(LDT_ENTRIES*LDT_ENTRY_SIZE);
          if (!ldt)
               printk(KERN_WARNING "ldt allocation failed\n");
          else
               memcpy(ldt, old_ldt, LDT_ENTRIES*LDT_ENTRY_SIZE);
     }
     new_mm->context.segments = ldt;
}
```

回到 copy_mm()的代码。对于 i386 CPU 来说，copy_mm()中 339 行处的 init_new_context()是个空语句。

当 CPU 从 copy_mm()回到 do_fork()中时，所有需要有条件复制的资源都已经处理完了。读者不妨回顾一下，当系统调用 fork()通过 sys_fork()进入 do_fork()时，其 clone_flags 为 SIGCHILD，也就是说，所有的标志位均为 0，所以 copy_files()、copy_fs()、copy_sighand()以及 copy_mm()全部都真正执行了，这四项资源全都复制了。而当 vfork()经过 sys_vfork()进入 do_fork()时，则其 clone_flags 为 VFORK | CLONE_VM | SIGCHLD，所以只执行了 copy_files()、copy_fs()以及 copy_sighand()；而 copy_mm()，则因为标志位 CLONE_VM 为 1，只是通过指针共享其父进程的 mm_struct，并没有一份自己的副本。这也就是说，经 vfork()复制的是个线程，只能靠共享其父进程的存储空间度日，包括用户空间堆栈在内。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

至于__clone()，则取决于调用时的参数。当然，最终还得取决于父进程具有什么资源，要是父进程没有已打开文件，那么即使执行了 copy_files()，也还是空的。

回到 do_fork()的代码中。前面已通过 alloc_task_struct()分配了两个连续的页面，其低端用作 task_struct 结构，已经基本上复制好了；而用作系统空间堆栈的高端，却还没有复制。现在就由 copy_thread()来做这件事了。这个函数的代码在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 523 552 ====================

```
523 /*
524  * Save a segment.
525  */
526 #define savesegment(seg,value) \
527     asm volatile("movl %%" #seg ",%0":"=m" (*(int *)&(value)))
528
529 int copy_thread(int nr, unsigned long clone_flags, unsigned long esp,
530     unsigned long unused,
531     struct task_struct * p, struct pt_regs * regs)
532 {
533     struct pt_regs * childregs;
534
535     childregs = ((struct pt_regs *) (THREAD_SIZE + (unsigned long) p)) - 1;
536     struct_cpy(childregs, regs);
537     childregs->eax = 0;
538     childregs->esp = esp;
539
540     p->thread.esp = (unsigned long) childregs;
541     p->thread.esp0 = (unsigned long) (childregs+1);
542
543     p->thread.eip = (unsigned long) ret_from_fork;
544
545     savesegment(fs,p->thread.fs);
546     savesegment(gs,p->thread.gs);
547
548     unlazy_fpu(current);
549     struct_cpy(&p->thread.i387, &current->thread.i387);
550
551     return 0;
552 }
```

名为 copy_thread()，实际上却只是复制父进程的系统空间堆栈。堆栈中的内容说明了父进程从通过系统调用进入系统空间开始到进入 copy_thread()的来历，子进程将要循相同的路线返回，所以要把它复制给子进程。但是，如果子进程的系统空间堆栈与父进程的完全相同，那返回以后就无从区分谁是子进程了，所以复制以后还要略作调整。这是一段很有趣的程序，我们先来看 535 行。在第 3 章中，读者已经看到当一个进程因系统调用或中断而进入内核时，其系统空间堆栈的顶部保存着 CPU 进入内核前夕各个寄存器的内容，并形成一个 pt_regs 数据结构。这里 535 行中的 p 为子进程的 task_struct 指针，指向两个连续物理页面的起始地址；而 THREAD_SIZE+(unsigned long)p 则指向这两个页面的顶端。将其变换成 struct pt_regs*，再从中减 1，就指向了子进程系统空间堆栈中的 pt_regs 结构，如图 4.3 所

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

示。

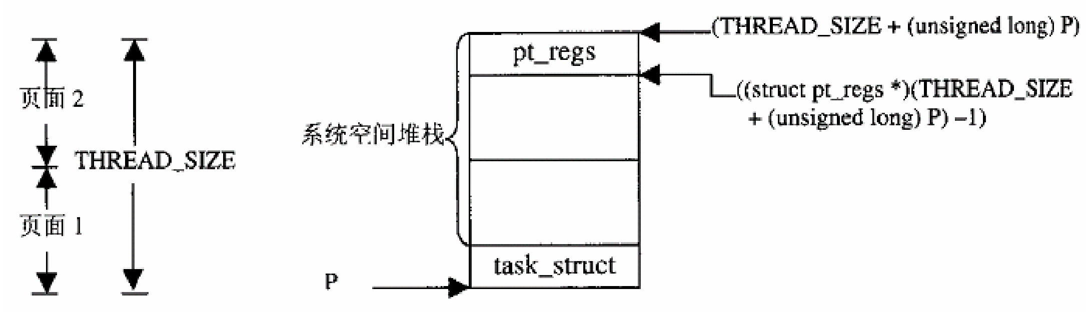


图 4.3 子进程系统空间堆栈示意图

得到了指向子进程系统空间堆栈中 pt_regs 结构的指针 childregs 以后，就先将当前进程系统空间堆栈中的 pt_regs 结构复制过去，再来作少量的调整。什么样的调整呢？首先，将该结构中的 eax 置成 0。当子进程受调度而“恢复”运行，从系统调用“返回”时，这就是返回值。如前所述，子进程的返回值为 0。其次，还要将结构中的 esp 置成这里的参数 esp，它决定了进程在用户空间的堆栈位置。在__clone()调用中，这个参数是由调用者给定的。而在 fork()和 vfork()中，则来自调用 do_fork()前夕的 regs.esp，所以实际上并没有改变，还是指向父进程原来在用户空间的堆栈。

在进程的 task_struct 结构中有个重要的成分 thread，它本身是一个数据结构 thread_struct，里面记录着进程在切换时的（系统空间）堆栈指针，取指令地址（也就是“返回地址”）等关键性的信息。在复制 task_struct 数据结构的时候，这些信息也原封不动地复制了过来。可是，子进程有自己的系统空间堆栈，所以也要相应加以调整。具体地说，540 行将 p->thread.esp 设置成子进程系统空间堆栈中 pt_regs 结构的起始地址，就好像这个子进程以前曾经运行过，而在进入内核以后正要返回用户空间时被切换了一样。而 p->thread.esp0 则应该指向子进程的系统空间堆栈的顶端。当一个进程被调度运行时，内核会将这个变量的值写入 TSS 的 esp0 字段，表示这个进程进入 0 级运行时其堆栈的位置。此外，p->thread.eip 的值表示当进程下一次被切换进入运行时的切入点，类似于函数调用或中断的返回地址。将此地址设置成 ret_from_fork，使创建的子进程在首次被调度运行时就从那儿开始，这一点以后在阅读有关进程切换的代码时还要讲到。545 行和 546 行的 savesegment 是个宏操作，其定义就在 526 行。所以，545 行在 gcc 预处理以后就会变成

```
asm volatile (“movl %%fs, %0 ” : ” = m” (* (int *) & p->thread.fs))
```

也就是把当前的段寄存器 fs 的值保存在 p->thread.fs 中。546 行与此类似。548 行和 549 行是为 i387 浮点处理器而设的，那就不是我们所关心的了。

回到 do_fork()，再往下看：

==================== kernel/fork.c 656 706 ====================
[sys_fork()>do_fork()]

```
	/* Our parent execution domain becomes current domain
	   These must match for thread signalling to apply */

	p->parent_exec_id = p->self_exec_id;
```

```

    /* ok, now we should be set up.. */
    p->swappable = 1;
    p->exit_signal = clone_flags & CSIGNAL;
    p->pdeath_signal = 0;

    /*
     * "share" dynamic priority between parent and child, thus the
     * total amount of dynamic priorities in the system doesnt change,
     * more scheduling fairness. This is only important in the first
     * timeslice, on the long run the scheduling behaviour is unchanged.
     */
    p->counter = (current->counter + 1) >> 1;
    current->counter >>= 1;
    if (!current->counter)
        current->need_resched = 1;

    /*
     * Ok, add it to the run-queues and make it
     * visible to the rest of the system.
     *
     * Let it rip!
     */
    retval = p->pid;
    p->tgid = retval;
    INIT_LIST_HEAD(&p->thread_group);
    write_lock_irq(&tasklist_lock);
    if (clone_flags & CLONE_THREAD) {
        p->tgid = current->tgid;
        list_add(&p->thread_group, &current->thread_group);
    }
    SET_LINKS(p);
    hash_pid(p);
    nr_threads++;
    write_unlock_irq(&tasklist_lock);

    if (p->ptrace & PT_PTRACED)
        send_sig(SIGSTOP, p, 1);

    wake_up_process(p);         /* do this last */
    ++total_forks;

fork_out:
    if ((clone_flags & CLONE_VFORK) && (retval > 0))
        down(&sem);
    return retval;

```

代码中的 parent_exec_id 表示父进程的执行域，self_exec_id 为本进程的执行域，swappable 表示本

进程的存储页面可以被换出，exit_signal 为本进程执行 exit()时应向父进程发出的信写，pdeath_signal 为要求父进程在执行 exit()时向本进程发出的信号。此外，task_struct 结构中 counter 字段的值就是进程的运行时间配额，这里将父进程的时间配额分成两半，让父、子进程各有原值的一半。如果创建的是线程，则还要通过 task_struct 结构中的队列头 thread_group 与父进程链接起来，形成一个“线程组”。接着，就要让子进程进入它的关系网了。先通过 SET_LINKS(p)将子进程的 task_struct 结构链入内核的进程队列，然后又通过 hash_pid()将其链入按其 pid 计算得的杂凑队列。有关这些队列的详情可参看“进程”以及“进程的调度与切换”两节中的有关叙述。最后，通过 wake_up_process()将子进程“唤醒”，也就是将其挂入可执行进程队列等待调度。有关详情可参看“过程的睡眠与唤醒”一节。

至此，新进程的创建已经完成了，并且已经挂入了可运行进程的队列接受调度。子进程与父进程在用户空间中具有相同的返回地址，然后才会因用户空间中程序的安排而分开。同时，由于当父进程（当前进程）从系统调用返回的前夕可能会接受调度，所以，到底谁会先返回到用户空间是不确定的。不过，一般而言，由于父、子进程适用相同的调度政策，而父进程在可执行进程队列中排在子进程前面，所以父进程先运行的可能较大。

还有一种特殊情况要考虑。当调用 do_fork()的参数中 CLONE_VFORK 标志位为 1 时，一定要保证让子进程先运行，一直到子进程通过系统调用 execve()执行一个新的可执行程序或者通过系统调用 exit()退出系统时，才可以恢复父进程的运行。为什么呢？这要从用户空间的复制或共享这个问题说起。前面读者已经看到，在创建子进程时，对于父进程的用户空间可以通过复制父进程的 mm_struct 及其下属的各个 vm_area_struct 数据结构，再加上父进程的页面目录和页面表来继承；也可以简单地复制父进程的 task_struct 结构中指向其 mm_struct 结构的指针来共享，具体取决于 CLONE_VM 标志位的值。当 CLONE_VM 标志位为 1，因而父、子进程通过指针共享用户空间时，父、子进程是在真正的意义上共享用户空间，父进程写入其用户空间的内容同时也“写入”子进程的用户空间，反之亦然。如果说，在这种情况下父、子进程各自对其数据区的写入可能会引起问题的话，那么对堆栈区的写入可就是致命的了。而每次对子程序的调用都是对堆栈区的写入！由此可见，在这样的情况下绝不能让两个进程都回到用户空间并发地运行；否则，必然是两个进程最终都乱来一气或者因非法越界访问而死亡。解决的办法只能是“扣留”其中一个进程，而只让一个进程回到用户空间，直到两个进程不再共享它们的用户空间或其中一个进程（必然是回到用户空间运行的那个进程）消亡为止。

所以，do_fork()中的 703 行和 704 行在 CLONE_VFORK 标志为 1 并且 fork 子进程成功的情况下，通过让当前进程（父进程）在一个信号量上执行一次 down()操作，以达到扣留父进程的目的。我们来看看具体是怎样实现的。

首先，信号量 sem 是在函数开头时的 560 行定义的一个局部量（名曰 DECLARE，实际上为之分配了空间）

==================== kernel/fork.c 560 560 ====================

[sys_fork()>do_fork()]

```
560         DECLARE_MUTEX_LOCKED(sem);
```

这儿 DECLARE_MUTEX_LOCKED 是在 include/asm-i386/semaphore.h 中定义的：

==================== include/asm-i386/semaphore.h 70 71 ====================

```
70   #define DECLARE_MUTEX(name) __DECLARE_SEMAPHORE_GENERIC(name,1)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
define DECLARE_MUTEX_LOCKED(name) __DECLARE_SEMAPHORE_GENERIC(name,0)
```

将 DECLARE_MUTEX_LOCKED 与 DECLARE_MUTEX 作一比较，可以看出正常情况下信号量中资源的数量为 1，而现在这个信号量中资源的数量为 0。当资源数量为 1 时，第一个执行 down()操作的进程进入临界区，而使资源数量变成了 0，以后执行加 down()操作的进程便会因为资源为 0 而被拒之门外进入睡眠，直到第一个进程归还资源离开临界区时才被唤醒。而现在这个信号量的资源从一开始就是 0，所以第一个对此执行 down()操作的进程就会进入睡眠，一直要到某个进程往这个信写量中投入资源，也就是执行一次 up()操作时才会被唤醒。

那么，谁来投入资源呢？在“系统调用 execve()”一节中读者将看到，子进程在通过 execve()执行一个新的可执行程序时会做这件事。此外，子进程在通过 exit()退出系统时也会做这件事。这里还要指出，这个信号量是 do_fork()的一个局部变量，所以在父进程的系统空间堆栈中，而子进程在其 task_struct 结构中有指向这个信号量的指针（即 vfork_sem，见 do_fork()的第 586 行）。既然父进程一直要睡眠到子进程使用这个信号量以后，信号量所在的空间就不会受到打扰。还应指出，CLONE_VM 要与 CLONE_VFORK 结合使用，否则就会发生前述的问题。除非在用户程序中采取了特殊的预防措施。

不管怎样，子进程的创建终于完成了，让我们祝福这新的生命！可是，如果子进程只具有与父进程相同的可执行程序和数据，只是父进程的“影子”，那又有什么意义呢？子进程必须走自己的路，这就是下一节“系统调用 execve()”所要讲述的内容了。

4.4 系统调用 execve()

读者在前一节中已经看到，进程通常是按其父进程的原样复制出来的，在多数情况下，如果复制出来的子进程不能与父进程分道扬镳，“走自己的路”，那就没有多大意义。所以，执行一个新的可执行程序是进程生命历程中关键性的一步。Linux 为此提供了一个系统调用 execve()，而在 C 语言的程序库中则又在此基础上向应用程序提供一整套的库函数，包括 execl()、execlp()、execle()、execleo()、execv()和 execvp()。此外，还有库函数 system()，也与 exccve()有关，不过 system()是 fork()、execve()、wait4()的组合。我们已经在本章第 2 节介绍过应用程序怎样调用 execve()，现在我们就来介绍 execve()的实现。

系统调用 execve()内核入口是 sys_execve()，代码见 arch/i386/kernel/process.c：

==================== arch/i386/kernel/process.c 722 740 ====================

```
/*
 * sys_execve() executes a new program.
 */
asmlinkage int sys_execve(struct pt_regs regs)
{
	int error;
	char * filename;

	filename = getname((char *) regs.ebx);
	error = PTR_ERR(filename);
	if (IS_ERR(filename))
		goto out;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
734      error = do_execve(filename, (char **) regs.ecx, (char **) regs.edx, &regs);
735      if (error == 0)
736           current->ptrace &= ~PT_DTRACE;
737      putname(filename);
738  out:
739      return error;
740  }
```

以前讲过，系统调用进入内核时，regs.ebx 中的内容为应用程序中调用相应库函数时的第一个参数。在本章第 2 节所举的例子中，这个参数为指向字符串“ /bin/echo”的指针。现在，指针存放在 regs.ebx 中，但字符串本身还在用户空间中，所以 730 行的 getname()要把这个字符串从用户空间拷贝到系统空间，在系统空间中建立起一个副本。让我们看看具体是怎么做的。函数 getname()的代码在 fs/namei.c 中：

==================== fs/namei.c 129 145 ====================

[sys_execve()>getname()]

```
129  char * getname(const char * filename)
130  {
131      char *tmp, *result;
132
133      result = ERR_PTR(-ENOMEM);
134      tmp = __getname();
135      if (tmp)  {
136           int retval = do_getname(filename, tmp);
137
138           result = tmp;
139           if (retval < 0) {
140                putname(tmp);
141                result = ERR_PTR(retval);
142           }
143      }
144      return result;
145  }
```

先通过__getname()分配一个物理页面作为缓冲区，然后调用 do_getname()从用户空间拷贝字符串。那么，为什么要专门为此分配一个物理页面作为缓冲区呢？首先，这个字符串确有可能相当长，因为这是一个绝对路径名。其次，我们以前讲过，进程系统空间堆栈的大小是大约 7KB，不能滥用，不宜在 getname()中定义一个局部的 4KB 的字符数组（注意，局部变量所占据的空间是在堆找中分配的）。函数 do_getname()的代码也在文件 fs/namei.c 中：

==================== fs/namei.c 102 127 ====================

[sys_execve()>getname()>do_getname()]

```
102  /* In order to reduce some races, while at the same time doing additional
103   * checking and hopefully speeding things up, we copy filenames to the
104   * kernel data space before using them..
105   *
106   * POSIX.1 2.4: an empty pathname is invalid (ENOENT).
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 */
static inline int do_getname(const char *filename, char *page)
{
	int retval;
	unsigned long len = PATH_MAX + 1;

	if ((unsigned long) filename >= TASK_SIZE) {
		if (!segment_eq(get_fs(), KERNEL_DS))
			return -EFAULT;
	} else if (TASK_SIZE - (unsigned long) filename < PAGE_SIZE)
		len = TASK_SIZE - (unsigned long) filename;

	retval = strncpy_from_user((char *)page, filename, len);
	if (retval > 0) {
		if (retval < len)
			return 0;
		return -ENAMETOOLONG;
	} else if (!retval)
		retval = -ENOENT;
	return retval;
}
```

如果指针 filename 的值大于等于 TASK_SIZE，就表示 filename 实际上在系统空间中。读者应该还记得 TASK_SIZE 的值是 3GB。具体的拷贝是通过 strncpy_from_user()进行的，代码见 arch/i386/lib/usercopy.c：

==================== arch/i386/lib/usercopy.c 100 107 ====================
[sys_execve()>getname()>do_getname()>strncpy_from_user()]

```
long
strncpy_from_user(char *dst, const char *src, long count)
{
	long res = -EFAULT;
	if (access_ok(VERIFY_READ, src, 1))
		__do_strncpy_from_user(dst, src, count, res);
	return res;
}
```

这个函数的主体 strncpy_from_user()是一个宏操作，也在同一源文件 arch/i386/lib/usercopy.c 中，与第 3 章中介绍过的__generic_copy_from_user()很相似，读者可以自行对照阅读。

在系统空间中建立起一份可执行文件的路径名副本以后，sys_execve()就调用 do_execve()，以完成其主体部分的工作。当然，完成以后还要通过 putname()将所分配的物理页面释放。函数 do_execve()的代码在 fs/exec.c 中，我们逐段地往下看：

==================== fs/exec.c 835 850 ====================
[sys_execve()>do_execve()]

```
/*
 * sys_execve() executes a new program.
 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs * regs)
{
	struct linux_binprm bprm;
	struct file *file;
	int retval;
	int i;

	file = open_exec(filename);

	retval = PTR_ERR(file);
	if (IS_ERR(file))
		return retval;

```

显然，先要将给定的可执行程序文件找到并打开，open_exec()就是为此而调用的，其代码也在 fs/exec.c 中，读者可结合“文件系统”一章中有关打开文件操作的内容，特别是 path_walk()的代码自行阅读。

假定目标文件已经打开，下一步就要从文件中装入可执行程序了。内核中为可执行程序的装入定义了一个数据结构 linux_binprm，以便将运行一个可执行文件时所需的信息组织在一起，这是在 include/linux/binfmts.h 定义的：

==================== include/linux/binfmts.h 19 33 ====================

```
/*
 * This structure is used to hold the arguments that are used when loading binaries.
 */
struct linux_binprm{
	char buf[BINPRM_BUF_SIZE];
	struct page *page[MAX_ARG_PAGES];
	unsigned long p; /* current top of mem */
	int sh_bang;
	struct file * file;
	int e_uid, e_gid;
	kernel_cap_t cap_inheritable, cap_permitted, cap_effective;
	int argc, envc;
	char * filename;	/* Name of binary */
	unsigned long loader, exec;
};
```

其中各个成分的作用读了下面的代码就会清楚。我们继续在 do_execve()中往下看：

==================== fs/exec.c 851 887 ====================

[sys_execve()>do_execve()]

```
	bprm.p = PAGE_SIZE*MAX_ARG_PAGES-sizeof(void *);
	memset(bprm.page, 0, MAX_ARG_PAGES*sizeof(bprm.page[0]));

	bprm.file = file;
	bprm.filename = filename;
	bprm.sh_bang = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    bprm.loader = 0;
    bprm.exec = 0;
    if ((bprm.argc = count(argv, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.argc;
    }

    if ((bprm.envc = count(envp, bprm.p / sizeof(void *))) < 0) {
        allow_write_access(file);
        fput(file);
        return bprm.envc;
    }

    retval = prepare_binprm(&bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings_kernel(1, &bprm.filename, &bprm);
    if (retval < 0)
        goto out;

    bprm.exec = bprm.p;
    retval = copy_strings(bprm.envc, envp, &bprm);
    if (retval < 0)
        goto out;

    retval = copy_strings(bprm.argc, argv, &bprm);
    if (retval < 0)
        goto out;

```

代码中的 linux_binprm 数据结构 bprm 是个局部量。函数 open_exec()返回一个 file 结构指针，代表着读入可执行文件的上下文，所以将其保存在数据结构 bprm 中。变量 bprm.sh_bang 的值说明可执行文件的性质，当可执行文件是一个 Shell 过程（Shell Script，用 Shell 语言编写的命令文件，由 shell 解释执行）时置为 1。而现在还不知道，所以暂且将其置为 0，也就是先假定为二进制文件。数据结构中的其它两个变量也暂时设置成 0。接着就处理可执行文件的参数和环境变量。

与可执行文件路径名的处理办法一样，每个参数的最大长度也定为一个物理页面，所以 bprm 中有一个页面指针数组，数组的大小为允许的最大参数个数 MAX_AGE_PAGES，目前这个常数定义为 32。前面已通过 memset()将这个指针数组初始化成全 0。现在将 bprm.p 设置成这些页面的总和减去一个指针的大小，因为第 0 个参数也就是 argv[0]是可执行程序本身的路径名。函数 count()是在 exec.c 中定义的，这里用它对字符串指针数组 argv[]中参数的个数进行计数，而 bprm.p/sizeof(void *)表示允许的最大值。同样，对作为参数传过来的环境变量也要通过 count()计数。注意这里的数组 argv[]和 envp[]是在用户空间而不在系统空间，所以计数的操作并不那么简单。函数 count()的代码在 fs/exec.c 中，它本身的代码很简单，但是引用的宏定义 get_user()却颇有些挑战性，值得一读。它也与第 3 章中介绍过的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

__generic_copy_from_user()相似，我们把它留给读者作为练习。有关的代码在 include/asm-i386/uaccess.h 和 arch/i386/lib/getuser.S 中，调用的路径为[count()>get_user()>_get_user()>_get_user_4()]。如果 count()失败，即返回负值，则要对目标文件执行一次 allow_write_access()。这个函数是与 deny_write_access()配对使用的，目的在于防止其它进程（可能在另一个 CPU 上运行）在读入可执行文件期间通过内存映射改变它的内容（详见“文件系统”以及系统调用 mmap()）。与其配对的 deny_write_access()是在打开可执行文件时在 open_exec()中调用的。

完成了对参数和环境变量的计数以后，do_execve()又调用 prepare_binprm()，进一步做数据结构 bprm 的准备工作，从可执行文件中读入开头的 128 个字节到 linux_binprm 结构 bprm 中的缓冲区。当然，在读之前还要先检验当前进程是否有这个权力，以及该文件是否有可执行属性。如果可执行文件具有“ set uid”特性则要作相应的设置。这个函数的代码也在 fs/exec.c 中。由于涉及文件操作的细节，我们建议读者在学习了“文件系统”以后再回过来自行阅读。此处先说明为什么只是先读 128 个字节。这是因为，不管目标文件是 elf 格式还是 a.out 格式，或者别的格式，在开头 128 个字节中都包括了关于可执行文件属性的必要而充分的信息。等一下读者就会看到这些信息的用途。

最后的准备工作就是把执行的参数，也就是 argv[]，以及运行的环境，也就是 envp[]，从用户空间拷贝到数据结构 bprm 中。其中的第 1 个参数 argv[0]就是可执行文件的路径名，已经在 bprm.filename 中了，所以用 copy_strings_kernel()从系统空间中拷贝，其它的就要用 copy_strings()从用户空间拷贝。

至此，所有的准备工作都已完成，所有必要的信息都已经搜集到了 linux_binprm 结构 bprm 中，接下来就要装入并运行目标程序了（fs/exec.c）：

==================== fs/exec.c 888 906 ====================

[sys_execve()>do_execve()]

```
	retval = search_binary_handler(&bprm,regs);
	if (retval >= 0)
		/* execve success */
		return retval;

out:
	/* Something went wrong, return the inode and free the argument pages*/
	allow_write_access(bprm.file);
	if (bprm.file)
		fput(bprm.file);

	for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
		struct page * page = bprm.page[i];
		if (page)
			__free_page(page);
	}

	return retval;
}
```

显然，这里的关键是 search_binary_handler()。在深入到这个函数内部之前，先介绍一个大概。内核中有一个队列，叫 formats，挂在此队列中的成员是代表着各种可执行文件格式的“代理人”，每个

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

成员只认识并且处理一种特定格式的可执行文件的运行。在前面的准备阶段中，已经从可执行文件头部读入了 128 个字节存放在 bprm 的缓冲区，而且运行所需的参数和环境变量也已经收集在 bprm 中。现在就由 formats 队列中的成员逐个来认领，谁要是辨认到了它所代表的可执行文件格式，运行的事就交给它。要是都不认识呢？那就根据文件头部的信息再找找看，是否有为此种格式设计，仍是作为可动态安装模块实现的“代理人”存在于文件系统中。如果有的话就把这模块安装进来并且将其挂入到 formats 队列中，然后让 formats 队列中的各个“代理人”再来试一次。

函数 search_binary_handler()的代码也在 fs/exec.c 中，其中有一段是专门针对 alpha 处理器的条件编译，在下列的代码中跳过了这段条件编译语句：

==================== fs/exec.c 747 754 ====================

[sys_execve()>do_execve()>search_binary_handler()]

```
/*
 * cycle the list of binary formats handler, until one recognizes the image
 */
int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
{
	int try,retval=0;
	struct linux_binfmt *fmt;
#ifdef __alpha__
```

==================== fs/exec.c 785 832 ====================

```
#endif
	for (try=0; try<2; try++) {
		read_lock(&binfmt_lock);
		for (fmt = formats ; fmt ; fmt = fmt->next) {
			int (*fn)(struct linux_binprm *, struct pt_regs *) = fmt->load_binary;
			if (!fn)
				continue;
			if (!try_inc_mod_count(fmt->module))
				continue;
			read_unlock(&binfmt_lock);
			retval = fn(bprm, regs);
			if (retval >= 0) {
				put_binfmt(fmt);
				allow_write_access(bprm->file);
				if (bprm->file)
					fput(bprm->file);
				bprm->file = NULL;
				current->did_exec = 1;
				return retval;
			}
			read_lock(&binfmt_lock);
			put_binfmt(fmt);
			if (retval != -ENOEXEC)
				break;
			if (!bprm->file) {
				read_unlock(&binfmt_lock);
				return retval;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                }
            }
            read_unlock(&binfmt_lock);
            if (retval != -ENOEXEC) {
                break;
#ifdef CONFIG_KMOD
            }else{
#define printable(c) (((c)=='\t') || ((c)=='\n') || (0x20<=(c) && (c)<=0x7e))
                char modname[20];
                if (printable(bprm->buf[0]) &&
                    printable(bprm->buf[1]) &&
                    printable(bprm->buf[2]) &&
                    printable(bprm->buf[3]))
                     break; /* -ENOEXEC */
                sprintf(modname, "binfmt-%04x", *(unsigned short *)(&bprm->buf[2]));
                request_module(modname);
#endif
            }
        }
        return retval;
}
```

程序中有两层嵌套的 for 循环。内层是对 formats 队列中的每个成员循环，让队列中的成员逐个试试它们的 load_binary()函数，看看能否对上号。如果对上了号，那就把目标文件装入并将其投入运行，再返回一个正数或 0。当 CPU 从系统调用退回时，该目标文件的执行就真正开始了。否则，如果不能辨识，或者在处理的过程中出了错，就返回一个负数。出错代码-ENOEXEC 表示只是对不上号，而并没有发生其它的错误，所以循环回去，让队列中的下一个成员再来试试。但是如果出了错而又并不是-ENOEXEC，那就表示对上了号但出了其它的错，这就不用再让其它的成员来试了。

内层循环结束以后，如果失败的原因是-ENOEXEC，就说明队列中所有的成员都不认识目标文件的格式。这时候，如果内核支持动态安装模块（取决于编译选项 CONFIG_KMOD），就根据目标文件的第 2 和第 3 个字节生成一个 binfmt 模块名，通过 request_module()试着将相应的模块装入（见本书“文件系统”和“设备驱动”两章中的有关内容）。外层的 for 循环共进行两次，正是为了在安装了模块以后再来试一次。

能在 Linux 系统上运行的可执行程序的开头几个字节，特别是开头 4 个字节，往往构成一个所谓的 magic number，如果把它拆开成字节，则往往又是说明文件格式的字符。例如，elf 格式的可执行文件的头四个字节为“0x7F”、“e”、“l”和“f”；而 java 的可执行文件头部四个字节则为“c”、“a”、“f”和“e”。如果可执行文件为 Shell 过程或 perl 文件，即第一行的格式为#! /bin/sh 或#! /usr/bin/perl，此时第一个字符为“#”，第二个字符为“!”，后面是相应解释程序的路径名。

数据结构 linux_binfmt 定义于 include/linux/binfmts.h 中，前面已经看到过了。结构中有三个函数指针，load_binary 用来装入可执行程序，load_shlib 用来装入动态安装的公用库程序，而 core_dump 的作用则不言自明。显然，这里最根本的是 load_binary。同时，如果不搞清具体的装载程序怎样工作，就很难对 execve()、进而对 Linux 进程的运行有深刻的理解。下面我们以 a.out 格式为例，讲述装入并启动执行目标程序的过程。其实，a.out 格式的可执行文件已经渐渐被淘汰了，取而代之的是 elf 格式。但

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

是，a.out 格式要简单得多，并且方便我们通过它来讲述目标程序的装载和投入运行的过程，所以从篇幅考虑我们选择了 a.out。读者在搞清了 a.out 格式的装载和投入运行过程以后，可以自行阅读有关 elf 格式的相关代码。

4.4.1 a.out 格式目标文件的装载和投入运行

与 a.out 格式可执行文件有关的代码都在 fs/binfmt_aout.c 中。先来看 a.out 格式的 linux_binfmt 数据结构，这个数据结构就是在 formats 队列中代表 a.out 格式的：

==================== fs/binfmt_aout.c 38 40 ====================

```
static struct linux_binfmt aout_format = {
	NULL, THIS_MODULE, load_aout_binary, load_aout_library, aout_core_dump, PAGE_SIZE
};
```

读者可以将它与前面的数据结构的类型定义相对照。装载和投入运行 a.out 格式目标文件的函数为 load_aout_binary()。可以想像，这是个比较复杂的过程，函数也比较大。我们还是老办法，一段一段往下看。其代码在 fs/binfmt_aout.c 中：

==================== fs/binfmt_aout.c 249 268 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
/*
 * These are the functions used to load a.out style executables and shared
 * libraries.  There is no binary dependent code anywhere else.
 */

static int load_aout_binary(struct linux_binprm * bprm, struct pt_regs * regs)
{
	struct exec ex;
	unsigned long error;
	unsigned long fd_offset;
	unsigned long rlim;
	int retval;

	ex = *((struct exec *) bprm->buf);		/* exec-header */
	if ((N_MAGIC(ex) != ZMAGIC && N_MAGIC(ex) != OMAGIC &&
	     N_MAGIC(ex) != QMAGIC && N_MAGIC(ex) != NMAGIC) ||
	    N_TRSIZE(ex) || N_DRSIZE(ex) ||
	    bprm->file->f_dentry->d_inode->i_size < ex.a_text+ex.a_data+N_SYMSIZE(ex)+N_TXTOFF(ex)) {
		return -ENOEXEC;
	}
```

首先是检查目标文件的格式，看看是否对上号。所有 a.out 格式可执行文件（二进制代码）的开头都应该是一个 exec 数据结构，这是在 include/asm-i386/a.out.h 中定义的：

==================== include/asm-i386/a.out.h 4 18 ====================

```
struct exec
```

```
{
  unsigned long a_info;      /* Use macros N_MAGIC, etc for access */
  unsigned a_text;      /* length of text, in bytes */
  unsigned a_data;      /* length of data, in bytes */
  unsigned a_bss;       /* length of uninitialized data area for file, in bytes */
  unsigned a_syms;      /* length of symbol table data in file, in bytes */
  unsigned a_entry;          /* start address */
  unsigned a_trsize;         /* length of relocation info for text, in bytes */
  unsigned a_drsize;         /* length of relocation info for data, in bytes */
};

#define N_TRSIZE(a)    ((a).a_trsize)
#define N_DRSIZE(a)    ((a).a_drsize)
#define N_SYMSIZE(a)   ((a).a_syms)
```

结构中的第一个无符号长整数 a_info 在逻辑上分成两部分：其高 16 位是一个代表目标 CPU 类型的代码，对于 i386 CPU 这部分的值为 100（0x64）；而低 16 位就是 magic number。不过，a.out 文件的 magic number 并不像在有的格式中那样是可打印字符，而是表示某些属性的编码，一共有四种，即 ZMAGIC、OMAGIC、QMAGIC 以及 NMAGIC，这是在 include/linux/a.out.h 中定义的：

==================== include/linux/a.out.h 60 71 ====================

```
/* Code indicating object file or impure executable.  */
#define OMAGIC 0407
/* Code indicating pure executable.  */
#define NMAGIC 0410
/* Code indicating demand-paged executable.  */
#define ZMAGIC 0413
/* This indicates a demand-paged executable with the header in the text.
   The first page is unmapped to help trap NULL pointer references */
#define QMAGIC 0314

/* Code indicating core file.  */
#define CMAGIC 0421
```

如果 magic number 不符，或者 exec 结构中提供的信息与实际不符，那就不能认为这个目标文件是 a.out 格式的，所以返回-ENOEXEC。

继续在 fs/binfmt_aout.c 中往下看：

==================== fs/binfmt_aout.c 270 287 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
	fd_offset = N_TXTOFF(ex);

	/* Check initial limits. This avoids letting people circumvent
	 * size limits imposed on them by creating programs with large
	 * arrays in the data or bss.
	 */
	rlim = current->rlim[RLIMIT_DATA].rlim_cur;
	if (rlim >= RLIM_INFINITY)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        rlim = ~0;
    if (ex.a_data + ex.a_bss > rlim)
        return -ENOMEM;

    /* Flush all traces of the currently running executable */
    retval = flush_old_exec(bprm);
    if (retval)
        return retval;

    /* OK, This is the point of no return */
```

各种 a.out 格式的文件因目标代码的特性不同，其正文的起始位置也就不同。为此提供了一个宏操作 N_TXTOFF()，以便根据代码的特性取得正文在目标文件中的起始位置，这是在 include/linux/a.out.h 中定义的：

==================== include/linux/a.out.h 80 86 ====================

```
#define _N_HDROFF(x) (1024 - sizeof (struct exec))

#if !defined (N_TXTOFF)
#define N_TXTOFF(x) \
 (N_MAGIC(x) == ZMAGIC ? _N_HDROFF((x)) + sizeof (struct exec) : \
  (N_MAGIC(x) == QMAGIC ? 0 : sizeof (struct exec)))
#endif
```

以前曾经讲过，每个进程的 task_struct 结构中有个数组 rlim，规定了该进程使用各种资源的限制，其中也包括对用于数据的内存空间的限制。所以，目标文件所确定的 data 和 bss 两个“段”的总和不能超出这个限制。

顺利通过了这些检验就表示具备了执行该目标文件的条件，所以就到了“与过去告别”的时候。这种“告别过去”意味着放弃从父进程“继承”下来的全部用户空间，不管是通过复制还是通过共享继承下来的。不过，下面读者会看到，这种告别也并非彻底的决裂。

函数 flush_old_exec()的代码也在 fs/exec.c 中：

==================== fs/exec.c 523 585 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()]

```
int flush_old_exec(struct linux_binprm * bprm)
{
    char * name;
    int i, ch, retval;
    struct signal_struct * oldsig;

    /*
     * Make sure we have a private signal table
     */
    oldsig = current->sig;
    retval = make_private_signals();
    if (retval) goto flush_failed;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /*
         * Release all of the old mmap stuff
         */
        retval = exec_mmap();
        if (retval) goto mmap_failed;

        /* This is the point of no return */
        release_old_signals(oldsig);

        current->sas_ss_sp = current->sas_ss_size = 0;

        if (current->euid == current->uid && current->egid == current->gid)
                current->dumpable = 1;
        name = bprm->filename;
        for (i=0; (ch = *(name++)) != '\0';) {
                if (ch == '/')
                        i = 0;
                else
                        if (i < 15)
                                current->comm[i++] = ch;
        }
        current->comm[i] = '\0';

        flush_thread();

        de_thread(current);

        if (bprm->e_uid != current->euid || bprm->e_gid != current->egid ||
              permission(bprm->file->f_dentry->d_inode,MAY_READ))
                current->dumpable = 0;

        /* An exec changes our domain. We are no longer part of the thread
           group */

        current->self_exec_id++;

        flush_signal_handlers(current);
        flush_old_files(current->files);

        return 0;

mmap_failed:
flush_failed:
        spin_lock_irq(&current->sigmask_lock);
        if (current->sig != oldsig)
                kfree(current->sig);
        current->sig = oldsig;
        spin_unlock_irq(&current->sigmask_lock);
        return retval;
```

```
}
```

首先是进程的信号（软中断）处理表。我们讲过，一个进程的信号处理表就好像一个系统中的中断向量表，虽然运用的层次不同，其概念是相似的。当子进程被创建出来时，父进程的信号处理表可能已经复制过来，但也有可能只是把父进程的信号处理表指针复制了过来，而通过这指针来共享父进程的信号处理表。现在，子进程最终要“自立门户”了，所以要看一下如果还在共享父进程的信号处理表的话，就要把它复制过来。正因为这样，make_private_signals()的代码与 do_fork()中调用的copy_sighand()基本相同。

==================== fs/exec.c 429 452 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>make_private_signals()]

```
/*
 * This function makes sure the current process has its own signal table,
 * so that flush_signal_handlers can later reset the handlers without
 * disturbing other processes.  (Other processes might share the signal
 * table via the CLONE_SIGNAL option to clone().)
 */

static inline int make_private_signals(void)
{
	struct signal_struct * newsig;

	if (atomic_read(&current->sig->count) <= 1)
		return 0;
	newsig = kmem_cache_alloc(sigact_cachep, GFP_KERNEL);
	if (newsig == NULL)
		return -ENOMEM;
	spin_lock_init(&newsig->siglock);
	atomic_set(&newsig->count, 1);
	memcpy(newsig->action, current->sig->action, sizeof(newsig->action));
	spin_lock_irq(&current->sigmask_lock);
	current->sig = newsig;
	spin_unlock_irq(&current->sigmask_lock);
	return 0;
}
```

读者也许要问：既然最终还是要把它复制过来，何不在当初一步就把它复制好了？这就是所谓“lazy computation”的概念：一件事情只有在非做不可时才做。虽然新创建的进程一般都会执行 execve()，“走自己的路”，但这是没有保证的。如果创建的是线程那就不一定会执行 execve()，如果一律在创建时就复制就可能造成浪费而且不符合要求。再说，检查一下是否还在与父进程共享信号处理表（通过检查共享计数）所花费的代价是很小的。当然，如果子进程是通过 fork()创建出来的话（而不是 vfork()或__clone()），那就一定都已经复制好了，这里的 make_private_signals()只不过是检查一下共享计数就马上回来了。

相比之下，exec_mmap()是更为关键的行动，从父进程继承下来的用户空间就是在这里放弃的。其代码在同一文件（fs/exec.c）中：

==================== fs/exec.c 385 427 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()]

```
static int exec_mmap(void)
{
	struct mm_struct * mm, * old_mm;

	old_mm = current->mm;
	if (old_mm && atomic_read(&old_mm->mm_users) == 1) {
		flush_cache_mm(old_mm);
		mm_release();
		exit_mmap(old_mm);
		flush_tlb_mm(old_mm);
		return 0;
	}

	mm = mm_alloc();
	if (mm) {
		struct mm_struct *active_mm = current->active_mm;

		if (init_new_context(current, mm)) {
			mmdrop(mm);
			return -ENOMEM;
		}

		/* Add it to the list of mm's */
		spin_lock(&mmlist_lock);
		list_add(&mm->mmlist, &init_mm.mmlist);
		spin_unlock(&mmlist_lock);

		task_lock(current);
		current->mm = mm;
		current->active_mm = mm;
		task_unlock(current);
		activate_mm(active_mm, mm);
		mm_release();
		if (old_mm) {
			if (active_mm != old_mm) BUG();
			mmput(old_mm);
			return 0;
		}
		mmdrop(active_mm);
		return 0;
	}
	return -ENOMEM;
}
```

同样，子进程的用户空间可能是父进程用户空间的复制品，也可能只是通过一个指针来共享父进程的用户空间，这一点只要检查一下对用户空间、也就是 current->mm 的共享计数就可清楚。当共享

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

计数为 1 时，表明对此空间的使用是独占的，也就是说这是从父进程复制过来的，那就要先释放 mm_struct 数据结构以下的所有 vm_area_struct 数据结构（但是不包括 mm_struct 结构本身），并且将页面表中的表项都设置成 0。具体地这是由 exit_mmap()完成的，其代码在 mm/mmap.c 中，读者可自行阅读。在调用 exit_mmap()之前还调用了一个函数 mm_release()，对此我们将在稍后加以讨论，因为在后面也调用了这个函数。至于 flush_cache_mm()和 flush_tlb_mm()，那只是使高速缓存与内存相一致，不在我们现在关心之列，而且前者对 i386 处理器而言根本就是空语句。这里倒是要问一句，在父进程 fork()子进程的时候，辛辛苦苦地复制了代表用户空间的所有数据结构，难道目的就在于稍后在执行 execve()时又辛辛苦苦把它们全部释放？既有今日，何必当初？是的，这确实不合理。这就是在有了 fork()系统调用以后又增加了一个 vfork()系统调用（从 BSD Unix 开始）的原因。让我们回顾一下 sys_fork()与 sys_vfork()在调用 do_fork()时的不同（arch/i386/kernel/process.c）：

==================== arch/i386/kernel/process.c 690 693 ====================

```
asmlinkage int sys_fork(struct pt_regs regs)
{
	return do_fork(SIGCHLD, regs.esp, &regs, 0);
}
```

==================== arch/i386/kernel/process.c 717 720 ====================

```
asmlinkage int sys_vfork(struct pt_regs regs)
{
	return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs, 0);
}
```

可见，sys_vfork()在调用 do_fork()时比 sys_fork()多了两个标志位，一个是 CLONE_VFORK，另一个是 CLONE_VM。当 CLONE_VM 标志位为 1 时，内核并不将父进程的用户空间（数据结构）复制给子进程，而只是将指向 mm_struct 数据结构的指针复制给子进程，让子进程通过这个指针来共享父进程的用户空间。这样，创建子进程时可以免去复制用户空间的麻烦。而当子进程调用 execve()就可以跳过释放用户空间这一步，直接就为子进程分配新的用户空间。但是，这样一来省事是省事了，却可能带来新的问题。以前讲过，fork()以后，execve()之前，子进程虽然有它自己的一整套代表用户空间的数据结构，但是最终在物理上还是与父进程共用相同的页面。不过，由于子进程有其独立的页面目录与页面表，可以在子进程的页面表里把对所有页面的访问权限都设置成“只读”。这样，当子进程企图改变某个页面的内容时，就会因权限不符而导致页面异常，在页面异常的处理程序中为子进程复制所需的物理页面，这就叫“ copy_on_write”。相比之下，如果子进程与父进程共享用户空间，也就是共享包括页面表在内的所有数据结构，那就无法实施“ copy_on_write”了。此时子进程所写入的内容就真正进入了父进程的空间中。我们知道，当一个进程在用户空间运行时，其堆栈也在用户空间。这意味着在这种情况下子进程可以改变父进程的堆栈，反过来父进程也可以改变子进程的堆栈！因为这个原因，vfork()的使用是很危险的，在子进程尚未放弃对父进程用户空间的共享之前，绝不能让两个进程都进入系统空间运行。所以，在 sys_vfork()调用 do_fork()时结合使用了另一个标志位 CLONE_VFORK。当这个标志位为 1 时，父进程在创建了子进程以后就进入睡眠状态，等候子进程通过 execve()执行另一个目标程序，或者通过 exit()寿终正寝。在这两种情况下子进程都会释放其共享的用户空间，使父进程可以安全地继续运行。即使如此，也还是有危险，子进程绝对不能从调用 vfork()的那个函数中返回，否

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

则还是可能破坏父进程的返回地址。所以，vfork()实际上是建立在子进程在创建以后立即就会调用execve()这个前提之上的。

那么，怎样使父进程进入睡眠而等待子进程调用 execve()或 exit()呢？当然可以有不同的实现。读者已经在 do_fork()的代码中看到了内核让父进程在一个 0 资源的“信号量”上执行一次 down()操作而进入睡眠的安排，这里的 mm_release()则让子进程在此信号量上执行一次 up()操作将父进程唤醒。函数 mm_release()的代码在 kernel/fork.c 中：

==================== kernel/fork.c 255 277 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()>mm_release()]

```
/* Please note the differences between mmput and mm_release.
 * mmput is called whenever we stop holding onto a mm_struct,
 * error success whatever.
 *
 * mm_release is called after a mm_struct has been removed
 * from the current process.
 *
 * This difference is important for error handling, when we
 * only half set up a mm_struct for a new process and need to restore
 * the old one.  Because we mmput the new mm_struct before
 * restoring the old one. . .
 * Eric Biederman 10 January 1998
 */
void mm_release(void)
{
	struct task_struct *tsk = current;

	/* notify parent sleeping on vfork() */
	if (tsk->flags & PF_VFORK) {
		tsk->flags &= ~PF_VFORK;
		up(tsk->p_opptr->vfork_sem);
	}
}
```

回到 exec_mmap()中，如果子进程的用户空间是通过指针共享而不是复制的，或者根本就没有用户空间，那就不需要调用 exit_mmap()释放代表用户空间的那些数据结构了。但是，此时要为子进程分配一个 mm_struct 数据结构以及页面目录，使得稍后可以在此基础上建立起子进程的用户空间。对于 i386 结构的 CPU，这里的 init_new_context()是空操作，永远返回 0，所以把它跳过。把当前进程的 task_struct 结构中的指针 mm 和 active_mm 设置成指向新分配的 mm_struct 数据结构以后，就要通过 activate_mm()切换到这个新的用户空间。这是一个宏操作，定义于 include/asm-i386/mmu_context.h：

==================== include/asm-i386/mmu_context.h 61 62 ====================

```
#define activate_mm(prev, next) \
	switch_mm((prev),(next),NULL,smp_processor_id())
```

我们将在“进程的调度与切换”一节中阅读 switch_mm()的代码，在这里只要知道当前进程的用户空间切换到了由新分配 mm_struct 数据结构所代表的空间就可以了。还要指出，现在新的“用户空间 ”

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

实际上只是一个框架，一个“空壳”，里面一个页面也没有。另一方面，现在是在内核中运行，所以用户空间的切换对目前的运行并无影响。

可是，原来的用户空间则从此与当前进程无关了。也就是说，当前进程最终放弃了对原来用户空间的共享。当然，此时要执行 mm_release()将父进程唤醒。实际上，CLONE_VFORK 通常都是与 CLONE_VM 标志相联系的，所以这里对 mm_release()的调用更为关键，而前面的 mm_release()则只是“以防万一”而已。那么，对于父进程的用户空间呢？当然要减少它的共享计数。此外，如果将它的共享计数减 1 以后达到了 0，则还要将其下属的数据结构释放，因为此时已没有进程还在使用这个空间了。这是由 mmput()完成的，其代码在 kernel/fork.c 中：

==================== kernel/fork.c 242 253 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()>mmput()]

```
/*
 * Decrement the use count and release all resources for an mm.
 */
void mmput(struct mm_struct *mm)
{
	if (atomic_dec_and_lock(&mm->mm_users, &mmlist_lock)) {
		list_del(&mm->mmlist);
		spin_unlock(&mmlist_lock);
		exit_mmap(mm);
		mmdrop(mm);
	}
}
```

就是说，将 mm->mm_users 减 1，如果减 1 以后变成了 0，就对 mm 执行 exit_mmap()和 mmdrop()。我们已经介绍过 exit_mmap()的作用，它释放 mm_struct 下面的所有 vm_area_struct 数据结构，并且将页面表中与用户空间相对应的表项都设置成 0，使整个“用户空间”成为了一个“空壳”。而 mmdrop()，则进一步将这个壳，也就是页面表和页面目录以及 mm_struct 数据结构本身，也全都释放了。不过，这只是在将父进程的 mm->mm_users 减 1 以后变成了 0 这种特殊情况下才发生。而在我们现在这个情景中，既然子进程通过指针共享父进程的用户空间，则父进程应该睡眠等待，所以当子进程释放对空间的共享时不会使共享计数达到 0。

回到前面 exec_mmap()的代码中，最后还有一个特殊情况要考虑，那就是当子进程进入 exec_mmap()时，其 task_struct 结构中的 mm_struct 结构指针 mm 为 0，也就是没有用户空间（所以是内核线程）。但是，另一个 mm_struct 结构指针 active_mm 却不为 0，这是因为在进程切换时的一个特殊要求而引起的。进程的 task_struct 中有两个 mm_struct 结构指针：一个是 mm，指向进程的用户空间，另一个是 active_mm。对于具有用户空间的进程这两个指针始终是一致的。但是，当一个不具备用户空间的进程（内核线程）被调度运行时，要求它的 active_mm 一定要指向某个 mm_struct 结构，所以只好暂借一个。在这种情况，内核将其 active_mm 设置成与在其之前运行的那个进程的 active_mm 相同，而在调度其停止运行时又将该指针设置成 0。也就是说，一个内核线程在受调度运行时要“借用”在它之前运行的那个进程的 active_mm（详见“进程的调度与切换”），因而要递增这个 mm_struct 结构的使用计数。而现在，已经为这内核线程分配了它自己的 mm_struct 结构，使其升格成为了进程，就不再使用借来的 active_mm 了。所以，要调用 mmdrop()，递减其使用计数。这是一个 inline 函数，其代码在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

include/linux/sched.h 中：

==================== include/linux/sched.h 709 715 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()>mmdrop()]

```
/* mmdrop drops the mm and the page tables */
extern inline void FASTCALL(__mmdrop(struct mm_struct *));
static inline void mmdrop(struct mm_struct * mm)
{
     if (atomic_dec_and_test(&mm->mm_count))
          __mmdrop(mm);
}
```

而__mmdrop()的代码则在 kernel/fork.c 中：

==================== kernel/fork.c 229 240 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()>mmdrop()>__mmdrop()]

```
/*
 * Called when the last reference to the mm
 * is dropped: either by a lazy thread or by
 * mmput. Free the page directory and the mm.
 */
inline void __mmdrop(struct mm_struct *mm)
{
     if (mm == &init_mm) BUG();
     pgd_free(mm->pgd);
     destroy_context(mm);
     free_mm(mm);
}
```

可见，mmdrop()在将一个 mm_struct 数据结构释放之前也要递减并检查其使用计数 mm_count，只有在递减后变成 0 才会将其释放。注意两个计数器，即 mm_users 与 mm_count 的区别。在 mm_struct 结构分配之初二者都设为 1，然后 mm_users 随子进程对用户空间的共享而增减，而 mm_count 则因内核中对该 mm_struct 数据结构的使用而增减。

从 exec_mmap()返回到 flush_old_exec()时，子进程从父进程继承的用户空间已经释放，其用户空间变成了一个独立的“空壳”，也就是一个大小为 0 的独立的用户空间。这时候的进程已经是“义无反顾”了，回不到原来的用户空间中去了（见代码中的注解）。前面讲过，当前进程（子进程）原来可能是通过指针共享父进程的信号处理表的，而现在有了自己的独立的信号处理表，所以也要递减父进程信号处理表的共享计数，并且如果递减后为 0 就要将其所占的空间释放，这就是 release_old_signals()所做的事情。此外，进程的 task_struct 结构中有一个字符数组 comm[]，用于保存进程所执行的程序名，所以还要把 bprm->filename 的目标程序路径名中的最后一段抄过去。接着的 flush_thread()只是处理与 debug 和 i387 协处理器有关的内容，不是我们所关心的。

如果“当前进程”原来只是一个线程，那么它的 task_struct 结构通过结构中的队列头 thread_group 挂入由其父进程为首的“线程组”队列。现在，它已经在通过 execve()升级为进程，放弃了对父进程用户空间的共享，所以就要通过 de_thread()从这个线程组中脱离出来。这个函数的代码在 fs/exec.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/exec.c 502 521 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>exec_mmap()>mmdrop()>__mmdrop()]

```
/*
 * An execve() will automatically "de-thread" the process.
 * Note: we don't have to hold the tasklist_lock to test
 * whether we migth need to do this. If we're not part of
 * a thread group, there is no way we can become one
 * dynamically. And if we are, we only need to protect the
 * unlink - even if we race with the last other thread exit,
 * at worst the list_del_init() might end up being a no-op.
 */
static inline void de_thread(struct task_struct *tsk)
{
     if (!list_empty(&tsk->thread_group)) {
          write_lock_irq(&tasklist_lock);
          list_del_init(&tsk->thread_group);
          write_unlock_irq(&tasklist_lock);
     }

     /* Minor oddity: this might stay the same. */
     tsk->tgid = tsk->pid;
}
```

前面说过，进程的信号处理表就好像是个中断向量表。但是，这里还有个重要的不同，就是中断向量表中的表项要么指向一个服务程序，要么就没有；而信号处理表中则还可以有对各种信号预设的（default）响应，并不一定非要指向一个服务程序。当把信号处理表从父进程复制过来时，其中每个表项的值有三种可能：一种可能是 SIG_IGN，表示不理睬；第二种是 SIG_DFL，表示采取预设的响应方式（例如收到 SIGQUIT 就 exit()）；第三种就是指向一个用户空间的子程序。可是，现在整个用户空间都已经放弃了，怎么还能让信号处理表的表项指向用户空间的子程序呢？所以还得检查一遍，将指向服务程序的表项改成 SIG_DFL。这是由 flush_signal_handlers()完成的，代码在 kernel/signal.c 中：

==================== kernel/signal.c 127 143 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>flush_signal_handlers()]

```
/*
 * Flush all handlers for a task.
 */

void
flush_signal_handlers(struct task_struct *t)
{
     int i;
     struct k_sigaction *ka = &t->sig->action[0];
     for (i = _NSIG ; i != 0 ; i--) {
          if (ka->sa.sa_handler != SIG_IGN)
               ka->sa.sa_handler = SIG_DFL;
          ka->sa.sa_flags = 0;
          sigemptyset(&ka->sa.sa_mask);
```

```
            ka++;
        }
}
```

最后，是对原有已打开文件的处理，这是由 flush_old_files()完成的。进程的 task_struct 结构中有个指向一个 file_struct 结构的指针 " files"，所指向的数据结构中保存着已打开文件的信息。在 file_struct 结构中有个位图 close_on_exec，里面存储着表示哪些文件在执行一个新目标程序时应予关闭的信息。而 flush_old_files()要做的就是根据这个位图的指示将这些文件关闭，并且将此位图清成全 0。其代码在 fs/exec.c 中：

==================== fs/exec.c 469 500 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>flush_old_exec()>flush_old_files()]

```
/*
 * These functions flushes out all traces of the currently running executable
 * so that a new one can be started
 */

static inline void flush_old_files(struct files_struct * files)
{
        long j = -1;

        write_lock(&files->file_lock);
        for (;;) {
                unsigned long set, i;

                j++;
                i = j * __NFDBITS;
                if (i >= files->max_fds || i >= files->max_fdset)
                        break;
                set = files->close_on_exec->fds_bits[j];
                if (!set)
                        continue;
                files->close_on_exec->fds_bits[j] = 0;
                write_unlock(&files->file_lock);
                for ( ; set ; i++,set >>= 1) {
                        if (set & 1) {
                                sys_close(i);
                        }
                }
                write_lock(&files->file_lock);

        }
        write_unlock(&files->file_lock);
}
```

一般来说，进程的开头三个文件，即 fd 为 0、1 和 2（或 stdin、stdout 以及 stderr）的已打开文件是不关闭的；其它的已打开文件则都应关闭，但是也可以通过 ioctl()系统调用来加以改变。

从 flush_old_exec()返回到 load_aout_binary()中时，当前进程已经完成了与过去告别，准备迎接新

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的使命了。我们继续沿着 fs/binfmt_aout.c 往下看（但是跳过针对 sparc 处理器的条件编译）:

==================== fs/binfmt_aout.c 287 307 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
	/* OK, This is the point of no return */
#if !defined(__sparc__)
	set_personality(PER_LINUX);
#else
	set_personality(PER_SUNOS);
#if !defined(__sparc_v9__)
	memcpy(&current->thread.core_exec, &ex, sizeof(struct exec));
#endif
#endif

	current->mm->end_code = ex.a_text +
		(current->mm->start_code = N_TXTADDR(ex));
	current->mm->end_data = ex.a_data +
		(current->mm->start_data = N_DATADDR(ex));
	current->mm->brk = ex.a_bss +
		(current->mm->start_brk = N_BSSADDR(ex));

	current->mm->rss = 0;
	current->mm->mmap = NULL;
	compute_creds(bprm);
	current->flags &= ~PF_FORKNOEXEC;
```

这里是对新的 mm_struct 数据结构中的一些变量进行初始化，为以后分配存储空间并读入可执行代码的映象作好准备。目标代码的映象分成 text、data 以及 bss 三段，mm_struct 结构中为每个段都设置了 start 和 end 两个指针。每段的起始地址定义于 include/linux/a.out.h:

==================== include/linux/a.out.h 108 111 ====================

```
/* Address of text segment in memory after it is loaded.  */
#if !defined (N_TXTADDR)
#define N_TXTADDR(x) (N_MAGIC(x) == QMAGIC ? PAGE_SIZE : 0)
#endif
```

==================== include/linux/a.out.h 141 154 ====================

```
#define _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))

#define _N_TXTENDADDR(x) (N_TXTADDR(x)+(x).a_text)

#ifndef N_DATADDR
#define N_DATADDR(x) \
    (N_MAGIC(x)==OMAGIC? (_N_TXTENDADDR(x)) \
     : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
#endif

/* Address of bss segment in memory after it is loaded.  */
#if !defined (N_BSSADDR)
#define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
```

```
#endif
```

可见，装入内存以后的程序映象从正文段（代码段）开始，其起始地址为 0 或 PAGE_SIZE，取决于具体的格式。正文段上面是数据段；然后是 bss 段，那就是不加初始化的数据段。再往上就是动态分配的内存“堆”以及用户空间的堆栈了。

然后，通过 compute_creds()确定进程在开始执行新的目标代码以后所具有的权限，这是根据 bprm 中的内容和当前的权限确定的。其代码在 exec.c 中，读者可自行阅读。

接下来，就取决于特殊 a.out 格式可执行代码的特性了（fs/binfmt_aout.c）：

==================== fs/binfmt_aout.c 308 309 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
#ifdef __sparc__
	if (N_MAGIC(ex) == NMAGIC) {
```

==================== fs/binfmt_aout.c 321 351 ====================

```
#endif

	if (N_MAGIC(ex) == OMAGIC) {
		unsigned long text_addr, map_size;
		loff_t pos;

		text_addr = N_TXTADDR(ex);

#if defined(__alpha__) || defined(__sparc__)
		pos = fd_offset;
		map_size = ex.a_text+ex.a_data + PAGE_SIZE - 1;
#else
		pos = 32;
		map_size = ex.a_text+ex.a_data;
#endif

		error = do_brk(text_addr & PAGE_MASK, map_size);
		if (error != (text_addr & PAGE_MASK)) {
			send_sig(SIGKILL, current, 0);
			return error;
		}

		error = bprm->file->f_op->read(bprm->file, (char *)text_addr,
			  ex.a_text+ex.a_data, &pos);
		if (error < 0) {
			send_sig(SIGKILL, current, 0);
			return error;
		}

		flush_icache_range(text_addr, text_addr+ex.a_text+ex.a_data);
	} else {
```

前面讲过，a.out 格式目标代码中的 magic number 表示着代码的特性，或者说类型。当 magic number 为 OMAGIC 时，表示该文件中的可执行代码并非“纯代码”。对于这样的代码，先通过 do_brk()为正

PDF-XCHANGE Click to buy NOW! www.docu-track.com

文段和数据段合在一起分配空间，然后就把这两部分从文件中读进来。函数 do_brk()我们已经在第 2 章中介绍过，而从文件读入则在“文件系统”和“块设备驱动”两章中有详细叙述，读者可以参阅，这里就不重复了。不过要指出，读入代码时是从文件中位移为 32 的地方开始，读入到进程用户空间中从地址 0 开始的地方，读入的总长度为 ex.a_text+ex.a_data。对于 i386 CPU 而言，flush_icache_range()为一空语句。至于 bss 段，则无需从文件读入，只要分配空间就可以了，所以放在后面再处理。对于 OMAGIC 类型的 a.out 可执行文件而言，装入程序的工作就基本完成了。

可是，如果不是 OMAGIC 类型呢？请接着往下看（fs/binfmt_aout.c）：

==================== fs/binfmt_aout.c 351 402 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
	} else {
		static unsigned long error_time, error_time2;
		if ((ex.a_text & 0xfff || ex.a_data & 0xfff) &&
		    (N_MAGIC(ex) != NMAGIC) && (jiffies-error_time2) > 5*HZ)
		{
			printk(KERN_NOTICE "executable not page aligned\n");
			error_time2 = jiffies;
		}

		if ((fd_offset & ~PAGE_MASK) != 0 &&
		    (jiffies-error_time) > 5*HZ)
		{
			printk(KERN_WARNING
			       "fd_offset is not page aligned. Please convert program: %s\n",
			       bprm->file->f_dentry->d_name.name);
			error_time = jiffies;
		}

		if (!bprm->file->f_op->mmap||((fd_offset & ~PAGE_MASK) != 0)) {
			loff_t pos = fd_offset;
			do_brk(N_TXTADDR(ex), ex.a_text+ex.a_data);
			bprm->file->f_op->read(bprm->file,(char *)N_TXTADDR(ex),
					ex.a_text+ex.a_data, &pos);
			flush_icache_range((unsigned long) N_TXTADDR(ex),
					   (unsigned long) N_TXTADDR(ex) +
					   ex.a_text+ex.a_data);
			goto beyond_if;
		}

		down(&current->mm->mmap_sem);
		error = do_mmap(bprm->file, N_TXTADDR(ex), ex.a_text,
			PROT_READ | PROT_EXEC,
			MAP_FIXED | MAP_PRIVATE | MAP_DENYWRITE | MAP_EXECUTABLE,
			fd_offset);
		up(&current->mm->mmap_sem);

		if (error != N_TXTADDR(ex)) {
```

```
            send_sig(SIGKILL, current, 0);
            return error;
        }

        down(&current->mm->mmap_sem);
        error = do_mmap(bprm->file, N_DATADDR(ex), ex.a_data,
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE | MAP_DENYWRITE | MAP_EXECUTABLE,
                fd_offset + ex.a_text);
        up(&current->mm->mmap_sem);
        if (error != N_DATADDR(ex)) {
            send_sig(SIGKILL, current, 0);
            return error;
        }
    }
```

在 a.out 格式的可执行文件中，除 OMAGIC 以外其它三种均为纯代码；也就是所谓的“可重入”代码。此类代码中，不但其正文段的执行代码在运行时不会改变，其数据段的内容也不会在运行时改变。凡是要在运行过程中改变内容的东西都在堆栈中（局部变量），要不然就在动态分配的缓冲区中。所以，内核干脆将可执行文件映射到了进程的用户空间中，这样连通常 swap 所需的盘上空间也省去了。在这三种类型的可执行文件中，除 NMAGIC 以外都要求正文段及数据段的长度与页面大小对齐。如发现没有对齐就要通过 printk()发出警告信息。但是，发出警告信息太频繁也不好，所以就设置了一个静态变量 error_time2，使警告信息之间的间隔不小于 5 秒。接下来的操作取决于具体的文件系统是否提供 mmap、就是将一个已打开文件映射到虚存空间的操作，以及正文段及数据段的长度是否与页面大小对齐。如果不满足映射的条件，就分配空间并且将正文段和数据段一起读入至进程的用户空间，这次是从文件中位移为 fd_offset，即 N_TXTOFF(ex)的地方开始，读入到由文件的头部所指定的地址 N_TXTADDR(ex)，长度为两段的总和。如果满足映射的条件，那就更好了，那就通过 do_mmap()分别将文件的正文段和数据段映射到进程的用户空间中，映射的地址则与装入的地址一致。调用 mmap()之前无需分配空间，那已经包含在 mmap()之中了。

至此，正文段和数据段都已经装入就绪了，接下来就是 bss 段和堆栈段了（fs/binfmt_aout.c）：

==================== fs/binfmt_aout.c 403 416 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
beyond_if:
    set_binfmt(&aout_format);

    set_brk(current->mm->start_brk, current->mm->brk);

    retval = setup_arg_pages(bprm);
    if (retval < 0) {
        /* Someone check-me: is this error path enough? */
        send_sig(SIGKILL, current, 0);
        return retval;
    }

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    current->mm->start_stack =
        (unsigned long) create_aout_tables((char *) bprm->p, bprm);
```

函数 set_binfmt()的操作很简单（fs/exec.c）：

==================== fs/exec.c 908 916 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>set_binfmt()]

```
void set_binfmt(struct linux_binfmt *new)
{
    struct linux_binfmt *old = current->binfmt;
    if (new && new->module)
        __MOD_INC_USE_COUNT(new->module);
    current->binfmt = new;
    if (old && old->module)
        __MOD_DEC_USE_COUNT(old->module);
}
```

如果当前进程原来执行的代码格式与新的代码格式都不是由可安装模块支持，则实际下只剩下一行语句，那就是设置 current->binfmt。

函数 set_brk()为可执行代码的 bss 段分配空间并建立起页面映射，其代码在同一文件中（fs/binfmt_aout.c）：

==================== fs/binfmt_aout.c 42 49 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>set_brk()]

```
 * dumping of the process results in another error..
 */

static int aout_core_dump(long signr, struct pt_regs * regs, struct file *file)
{
    mm_segment_t fs;
    int has_dumped = 0;
    unsigned long dump_start, dump_size;
```

读者在第 2 章中读过 do_brk()的代码，应该理解为什么 bss 段中内容的初始值为全 0。

接着，还要在用户空间的堆栈区顶部为进程建立起一个虚存区间，并将执行参数以及环境变量所占的物理页面与此虚存区间建立起映射。这是由 setup_arg_pages()完成的，其代码在 fs/exec.c 中：

==================== fs/exec.c 288 332 ====================
[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>setup_arg_pages()]

```
int setup_arg_pages(struct linux_binprm *bprm)
{
    unsigned long stack_base;
    struct vm_area_struct *mpnt;
    int i;

    stack_base = STACK_TOP - MAX_ARG_PAGES*PAGE_SIZE;

    bprm->p += stack_base;
    if (bprm->loader)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        bprm->loader += stack_base;
    bprm->exec += stack_base;

    mpnt = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!mpnt)
        return -ENOMEM;

    down(&current->mm->mmap_sem);
    {
        mpnt->vm_mm = current->mm;
        mpnt->vm_start = PAGE_MASK & (unsigned long) bprm->p;
        mpnt->vm_end = STACK_TOP;
        mpnt->vm_page_prot = PAGE_COPY;
        mpnt->vm_flags = VM_STACK_FLAGS;
        mpnt->vm_ops = NULL;
        mpnt->vm_pgoff = 0;
        mpnt->vm_file = NULL;
        mpnt->vm_private_data = (void *) 0;
        insert_vm_struct(current->mm, mpnt);
        current->mm->total_vm = (mpnt->vm_end - mpnt->vm_start) >> PAGE_SHIFT;
    }

    for (i = 0 ; i < MAX_ARG_PAGES ; i++) {
        struct page *page = bprm->page[i];
        if (page) {
            bprm->page[i] = NULL;
            current->mm->rss++;
            put_dirty_page(current,page,stack_base);
        }
        stack_base += PAGE_SIZE;
    }
    up(&current->mm->mmap_sem);

    return 0;
}
```

进程的用户空间中地址最高处为堆栈区，这里的常数 STACK_TOP 就是 TASK_SIZE，也就是 3GB（0xC0000000）。堆栈区的顶部为一个数组，数组中的每一个元素都是一个页面。数组的大小为 MAX_ARG_PAGES，而实际映射的页面数量则取决于这些执行参数和环境变量的数量。

然后，在这些页面的下方，就是过程的用户空间堆栈了。另一方面，大家知道任何用户程序的入口都是 main()，而 main 有两个参数 argc 和 argv[]。其中参数 argv[]是字符指针数组，argc 则为数组的大小。但是实际上还有个隐藏着的字符指针数组 envp[]用来传递环境变量，只是不在用户程序的“视野”之内而已。所以，用户空间堆栈中从一开始就要设置好三项数据，即 envp[]、argv[]以及 argc。此外，还要将保存着的（字符串形式的）参数和环境变量复制到用户空间的顶端。这都是由 create_aout_tables()完成的，其代码也在同一文件（fs/binfmt_aout.c）中：

==================== fs/binfmt_aout.c 187 200 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()>create_aout_table()]

```
/*
 * create_aout_tables() parses the env- and arg-strings in new user
 * memory and creates the pointer tables from them, and puts their
 * addresses on the "stack", returning the new stack pointer value.
 */
static unsigned long * create_aout_tables(char * p, struct linux_binprm * bprm)
{
        char **argv, **envp;
        unsigned long * sp;
        int argc = bprm->argc;
        int envc = bprm->envc;

        sp = (unsigned long *) ((-(unsigned long)sizeof(char *)) & (unsigned long) p);
#ifdef __sparc__
```

===================== fs/binfmt_aout.c 204 205 =====================

```
#endif
#ifdef __alpha__
```

===================== fs/binfmt_aout.c 217 247 =====================

```
#endif
        sp -= envc+1;
        envp = (char **) sp;
        sp -= argc+1;
        argv = (char **) sp;
#if defined(__i386__) || defined(__mc68000__) || defined(__arm__)
        put_user((unsigned long) envp,--sp);
        put_user((unsigned long) argv,--sp);
#endif
        put_user(argc,--sp);
        current->mm->arg_start = (unsigned long) p;
        while (argc-->0) {
                char c;
                put_user(p,argv++);
                do {
                        get_user(c,p++);
                } while (c);
        }
        put_user(NULL,argv);
        current->mm->arg_end = current->mm->env_start = (unsigned long) p;
        while (envc-->0) {
                char c;
                put_user(p,envp++);
                do {
                        get_user(c,p++);
                } while (c);
        }
        put_user(NULL,envp);
        current->mm->env_end = (unsigned long) p;
        return sp;
```

```
247 }
```

读者应该能看明白，这是在堆栈的顶端构筑 envp[]、argv[]和 argc。请读者注意看一下这段代码中的 228 至 234 行（以及 237 至 243 行），然后回答一个问题：为什么是 get_user(c, ptt)而不是 get_user(&c, ptt)？以前我们曾经讲过，get_user()是一段颇具挑战性的代码，并建议读者自行阅读。现在简要地介绍一下，看看你是否读懂了。这是在 include/asm-i386/uaccess.h 中定义的一个宏定义：

==================== include/asm-i386/uaccess.h 89 124 ====================

```
89   /*
90    * These are the main single-value transfer routines.  They automatically
91    * use the right size if we just have the right pointer type.
92    *
93    * This gets kind of ugly. We want to return _two_ values in "get_user()"
94    * and yet we don't want to do any pointers, because that is too much
95    * of a performance impact. Thus we have a few rather ugly macros here,
96    * and hide all the uglyness from the user.
97    *
98    * The "__xxx" versions of the user access functions are versions that
99    * do not verify the address space, that must have been done previously
100   * with a separate "access_ok()" call (this is used when we do multiple
101   * accesses to the same area of user memory).
102   */
103
104  extern void __get_user_1(void);
105  extern void __get_user_2(void);
106  extern void __get_user_4(void);
107
108  #define __get_user_x(size,ret,x,ptr) \
109       __asm__ __volatile__("call __get_user_" #size \
110            :"=a" (ret),"=d" (x) \
111            :"0" (ptr))
112
113  /* Careful: we have to cast the result to the type of the pointer for sign reasons */
114  #define get_user(x,ptr)                                       \
115  ({   int __ret_gu,__val_gu;                                   \
116       switch(sizeof (*(ptr))) {                               \
117       case 1:  __get_user_x(1,__ret_gu,__val_gu,ptr); break;       \
118       case 2:  __get_user_x(2,__ret_gu,__val_gu,ptr); break;       \
119       case 4:  __get_user_x(4,__ret_gu,__val_gu,ptr); break;       \
120       default: __get_user_x(X,__ret_gu,__val_gu,ptr); break;       \
121       }                                             \
122       (x) = (__typeof__(*(ptr)))__val_gu;                     \
123       __ret_gu;                                     \
124  })
```

先看一下 122 行，它回答了为什么引用时的第一个参数是 c 而不是&c 的问题。其次，经过 gcc 的预处理以后，__get_user_x()就变成__get_user_1()，__get_user_2()或__get_user_4()，分别用于从用户空间读取一个字节、一个短整数或一个长整数。宏操作 get_user 根据第 2 个参数的类型确定目标的大小

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

而分别调用__get_user_1()，__get_user_2()或__get_user_4()。调用时目标地址（ptt）在寄存器 EAX 中；而返回时 EAX 中为返回的函数值（出错代码），EDX 中为从用户空间读过来的数值。这几个函数的代码都在 arch/i386/lib/getuser.S 中，以__get_user_1()为例：

==================== arch/i386/lib/getuser.S 24 36 ====================

```
24   addr_limit = 12
25
26   .text
27   .align 4
28   .globl __get_user_1
29   __get_user_1:
30        movl %esp,%edx
31        andl $0xffffe000,%edx
32        cmpl addr_limit(%edx),%eax
33        jae bad_get_user
34   1:   movzbl (%eax),%edx
35        xorl %eax,%eax
36        ret
```

==================== arch/i386/lib/getuser.S 64 68 ====================

```
64   bad_get_user:
65        xorl %edx,%edx
66        movl $-14,%eax
67        ret
68
```

这里的第 30 和 31 行将当前进程的系统空间堆栈指针与 8K，即两个页面的边界对齐，从而取得当前进程的 task_struct 结构指针。在 task_struct 结构中位移 12 处为当前进程用户空间地址的上限，所以作为参数传过来的地址不得高于这个上限。这也说明，对 task_struct 结构的定义（开头几个成分）是不能随意更改的。如果地址没有超出范围就从用户空间把其内容读入寄存器 DX，并将 EAX 清 0 作为返回的函数值。

另一个宏操作 put_user()与此相似，只是方向相反。

当 CPU 从 create_aout_tables()返回到 do_load_aout_binary()时，堆栈顶端的 argv[]和 argc 都已经准备好。我们再继续往下看（fs/binfmt_aout.c）：

==================== fs/binfmt_aout.c 417 424 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_aout_binary()]

```
#ifdef __alpha__
     regs->gp = ex.a_gpvalue;
#endif
     start_thread(regs, ex.a_entry, current->mm->start_stack);
     if (current->ptrace & PT_PTRACED)
          send_sig(SIGTRAP, current, 0);
     return 0;
}
```

这里只剩下最后一个关键性的操作了，那就是 start_thread()。这是个宏操作，定义于 include/asm-i386/processor.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/asm-i386/processor.h 408 417 ====================

```
#define start_thread(regs, new_eip, new_esp) do {       \
        __asm__("movl %0,%%fs ; movl %0,%%gs": :"r" (0));   \
        set_fs(USER_DS);                        \
        regs->xds = __USER_DS;                  \
        regs->xes = __USER_DS;                  \
        regs->xss = __USER_DS;                  \
        regs->xcs = __USER_CS;                  \
        regs->eip = new_eip;                    \
        regs->esp = new_esp;                    \
} while (0)
```

读者对这里的 regs 指针已经很熟悉，它指向保留在当前进程系统空间堆栈中的各个寄存器副本。当进程从系统调用返回时，这些数值就会被“恢复”到 CPU 的各个寄存器中。所以，那时候的堆栈指针将是 current->mm->start_stack；而返回地址，也就是 EIP 的内容，则将是 ex.a_entry。显然，这正是我们所需要的。

至此，可执行代码的装入和投入运行已经完成。而 do_execve()在调用了 search_binary_handler()以后也就结束了。当 CPU 从系统调用返回到用户空间时，就会从由 ex.a_entry 确定的地址开始执行。

4.4.2 文字形式可执行文件的执行

前面介绍了 a.out 格式可执行文件的装入和投入运行过程，我们把这作为二进制可执行文件的代表。现在，再来简要地看一下字符形式的可执行文件（为 shell 过程或 perl 文件）的执行。有关的代码都在 binfmt_script.c 中。由于已经比较详细地阅读了二进制可执行文件的处理，读者在阅读下面的代码时应该比较轻松了，所以我们只作一些简要的提示（fs/binfmt_script.c）：

==================== fs/binfmt_script.c 95 97 ====================

```
struct linux_binfmt script_format = {
        NULL, THIS_MODULE, load_script, NULL, NULL, 0
};
```

以前我们提到过，Script 文件的开头的个字符应为“ #!”，然后是解释程序的路径名，如/bin/sh，/usr/bin/perl 等等，后面还可以有参数。但是，第一行的长度不得长于 127 个字符。我们来看 Script 文件的装载，这是由 load_script()完成的（fs/binfmt_script.c）：

==================== fs/binfmt_script.c 17 58 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_script()]

```
static int load_script(struct linux_binprm *bprm,struct pt_regs *regs)
{
        char *cp, *i_name, *i_arg;
        struct file *file;
        char interp[BINPRM_BUF_SIZE];
        int retval;

        if ((bprm->buf[0] != '#') || (bprm->buf[1] != '!') || (bprm->sh_bang))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            return -ENOEXEC;
        /*
         * This section does the #! interpretation.
         * Sorta complicated, but hopefully it will work.  -TYT
         */

        bprm->sh_bang++;
        allow_write_access(bprm->file);
        fput(bprm->file);
        bprm->file = NULL;

        bprm->buf[BINPRM_BUF_SIZE - 1] = '\0';
        if ((cp = strchr(bprm->buf, '\n')) == NULL)
            cp = bprm->buf+BINPRM_BUF_SIZE-1;
        *cp = '\0';
        while (cp > bprm->buf) {
            cp--;
            if ((*cp == ' ') || (*cp == '\t'))
                *cp = '\0';
            else
                break;
        }
        for (cp = bprm->buf+2; (*cp == ' ') || (*cp == '\t'); cp++);
        if (*cp == '\0')
            return -ENOEXEC; /* No interpreter name found */
        i_name = cp;
        i_arg = 0;
        for ( ; *cp && (*cp != ' ') && (*cp != '\t'); cp++)
            /* nothing */ ;
        while ((*cp == ' ') || (*cp == '\t'))
            *cp++ = '\0';
        if (*cp)
            i_arg = cp;
        strcpy (interp, i_name);
```

得到了解释程序的路径名以后，问题就转化成了对解释程序的装入，而 script 文件本身则转化成了解释程序的运行参数。虽然 script 文件本身并不是二进制格式的可执行文件，解释程序的映象却是一个二进制的可执行文件。还是在 fs/binfmt_script.c 文件中往下看：

==================== fs/binfmt_script.c 59 93 ====================

[sys_execve()>do_execve()>search_binary_handler()>load_script()]

```
        /*
         * OK, we've parsed out the interpreter name and
         * (optional) argument.
         * Splice in (1) the interpreter's name for argv[0]
         *           (2) (optional) argument to interpreter
         *           (3) filename of shell script (replace argv[0])
         *
         * This is done in reverse order, because of how the
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     * user environment and arguments are stored.
     */
    remove_arg_zero(bprm);
    retval = copy_strings_kernel(1, &bprm->filename, bprm);
    if (retval < 0) return retval;
    bprm->argc++;
    if (i_arg) {
        retval = copy_strings_kernel(1, &i_arg, bprm);
        if (retval < 0) return retval;
        bprm->argc++;
    }
    retval = copy_strings_kernel(1, &i_name, bprm);
    if (retval) return retval;
    bprm->argc++;
    /*
     * OK, now restart the process with the interpreter's dentry.
     */
    file = open_exec(interp);
    if (IS_ERR(file))
        return PTR_ERR(file);

    bprm->file = file;
    retval = prepare_binprm(bprm);
    if (retval < 0)
        return retval;
    return search_binary_handler(bprm,regs);
}
```

可见，Script 文件的使用在装入运行的过程中引入了递归性，load_script()最后又调用 search_binary_handler()。不管递归有多深，最终执行的一定是个二进制可执行文件，例如/bin/sh、/usr/bin/perl 等解释程序。在递归的过程中，逐层的可执行文件路径名形成一个参数堆栈，传递给最终的解释程序。

4.5 系统调用 exit()与 wait4()

系统调用 exit()与 wait4()的代码基本上都在 kernel/exit.c 中，下面我们在引用代码时凡不特别说明出处的均来自这个文件。

先来看 exit()的实现（kernel/exit.c）：

==================== kernel/exit.c 482 485 ====================

```
asmlinkage long sys_exit(int error_code)
{
    do_exit((error_code&0xff)<<8);
}
```

显然，其主体为 do_exit()。先看它的前半部：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== kernel/exit.c 421 433 ====================

[sys_exit()>do_exit()]

```
NORET_TYPE void do_exit(long code)
{
	struct task_struct *tsk = current;

	if (in_interrupt())
		panic("Aiee, killing interrupt handler!");
	if (!tsk->pid)
		panic("Attempted to kill the idle task!");
	if (tsk->pid == 1)
		panic("Attempted to kill init!");
	tsk->flags |= PF_EXITING;
	del_timer_sync(&tsk->real_timer);

```

首先，在函数的类型 void 前面还有个说明 NORET_TYPE。在 include/linux/kernel.h 中 NORET_TYPE 定义为 " /* */"，所以对编译毫无影响，但起到了提醒读者的作用。CPU 在进入 do_exit()以后，当前进程就在中途寿终正寝，不会从这个函数返回。所谓不会从这个函数返回到底是怎么回事，又是什么原因，读者在读了下面的代码以后就明白了。这里只指出，既然 CPU 不会从 do_exit()中返回，也就不会从 sys_exit()中返回，从而也就不会从系统调用 exit()返回。也只有这样，才能达到 " exit"，即从系统退出的目的。另一方面，所谓 exit，只有进程（或线程）才谈得上。中断服务程序根本就不应该调用 do_exit()，不管是直接还是间接调用。所以，这里首先通过 in_interrupt()对此加以检查，如发现这是在某个中断服务程序中调用的，那就一定是出了问题。

那么，怎么知道是否在中断服务程序中呢？让我们来看看在 include/asm-i386/hardirq.h 中定义的 in_interrupt()：

==================== include/asm-i386/hardirq.h 20 25 ====================

```
/*
 * Are we in an interrupt context? Either doing bottom half
 * or hardware interrupt processing?
 */
#define in_interrupt() ({ int __cpu = smp_processor_id(); \
	(local_irq_count(__cpu) + local_bh_count(__cpu) != 0); })
```

在单 CPU 的系统中，__cpu 一定是 0。在第 3 章中讲到过函数 handle_IRQ_event()，在其入口处和出口处各有一个函数调用 irq_enter()和 irq_exit()，就分别递增和递减计数器 local_irq_count[__cpu]。所以，只要这个计数器为非 0，就说明 CPU 在 handle_IRQ_event()中。类似地，只要计数器 local_bh_count[__cpu]为非 0，就说明 CPU 正在执行某个 bh 函数，这也跟中断服务程序一样。反之，只要不是在中断服务的上下文中，那就一定是在某个进程（或线铁）的上下文中了。但是，0 号进程和 1 号进程，也就是"空转"（idle）进程和"初始化"（init）进程，是不允许退出的，所以接着要对当前进程的 pid 加以检查。

进程在决定退出之前可能已经设置了实时定时器，也就是将其 task_struct 结构中的成员 real_timer 挂入了内核中的定时器队列。现在进程即将退出系统，一来是这个定时器已经没有了存在的必要，二

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

来进程的 task_struct 结构行将撤销，作为其成员的 real_timer 也将“皮之不存，毛将焉附”，当然要先将它从队列中脱离。所以，要通过 del_timer_sync()将当前进程从定时器队列中脱离出来。

继续往下看（kernel/exit.c）：

==================== kernel/exit.c 434 472 ====================

[sys_exit()>do_exit()]

```
fake_volatile:
#ifdef CONFIG_BSD_PROCESS_ACCT
	acct_process(code);
#endif
	__exit_mm(tsk);

	lock_kernel();
	sem_exit();
	__exit_files(tsk);
	__exit_fs(tsk);
	exit_sighand(tsk);
	exit_thread();

	if (current->leader)
		disassociate_ctty(1);

	put_exec_domain(tsk->exec_domain);
	if (tsk->binfmt && tsk->binfmt->module)
		__MOD_DEC_USE_COUNT(tsk->binfmt->module);

	tsk->exit_code = code;
	exit_notify();
	schedule();
	BUG();
/*
 * In order to get rid of the "volatile function does return" message
 * I did this little loop that confuses gcc to think do_exit really
 * is volatile. In fact it's schedule() that is volatile in some
 * circumstances: when current->state = ZOMBIE, schedule() never
 * returns.
 *
 * In fact the natural way to do all this is to have the label and the
 * goto right after each other, but I put the fake_volatile label at
 * the start of the function just in case something /really/ bad
 * happens, and the schedule returns. This way we can try again. I'm
 * not paranoid: it's just that everybody is out to get me.
 */
	goto fake_volatile;
}
```

可想而知，进程在结束生命退出系统之前要释放其所有的资源。我们在前一节的 do_fork()中看到从父进程“继承”的资源有存储空间、已打开文件、工作目录、信号处理表等等。相应地，这里就有

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

__exit_mm()、__exit_files()、__exit_fs()以及__exit_sighand()。可是，还有一种资源是不“继承”的，所以在 do_fork()中不会看到，那就是进程在用户空间建立和使用的“信号量”（semaphore）。这是一种用于进程间通讯的资源，如果在调用 exit()之前还有信号量尚未撤销，那就也要把它撤销。这里有一个简单的准则，就是看 task_struct 数据结构中的各个成分，如果一个成分是个指针，在进程创建时以及运行过程中要为其在内核中分配一个数据结构或缓冲区，而且这个指针又是通向这个数据结构或缓冲区的唯一途径，那就一定要把它释放，否则就会造成内核的存储空间“泄漏”。例如，指针 sig 指向进程的信号处理表，这个表所占的空间是专为 sig 分配的，指针 sig 就是进入这个表的唯一途径，所以必须释放。而指针 p_pptr 指向父进程的 task_struct 结构，可是父进程的 task_struct 结构却并不是专门为子进程的 p_pptr 而分配的，这个 p_pptr 并不是进入其父进程的 task_struct 的唯一途径，所以不能把这个数据结构也释放掉，否则其它指向这个结构的指针就都“悬空”了。具体到用户空间信号量，当进程在用户空间创建和使用信号量时，内核会为进程 task_struct 结构中的两个指针 semundo 和 semsleeping 分配缓冲区（sem_undo 数据结构和 sem_queue 数据结构，详见“进程间通信”）。而且，这两个指针就是进入这些数据结构的唯一途径，所以必须把它们释放。函数 sem_exit()的代码在 ipc/sem.c 中：

==================== ipc/sem.c 966 1041 ====================

[sys_exit()>do_exit()>sem_exit()]

```
/*
 * add semadj values to semaphores, free undo structures.
 * undo structures are not freed when semaphore arrays are destroyed
 * so some of them may be out of date.
 * IMPLEMENTATION NOTE: There is some confusion over whether the
 * set of adjustments that needs to be done should be done in an atomic
 * manner or not. That is, if we are attempting to decrement the semval
 * should we queue up and wait until we can do so legally?
 * The original implementation attempted to do this (queue and wait).
 * The current implementation does not do so. The POSIX standard
 * and SVID should be consulted to determine what behavior is mandated.
 */
void sem_exit (void)
{
     struct sem_queue *q;
     struct sem_undo *u, *un = NULL, **up, **unp;
     struct sem_array *sma;
     int nsems, i;

     /* If the current process was sleeping for a semaphore,
      * remove it from the queue.
      */
     if ((q = current->semsleeping)) {
          int semid = q->id;
          sma = sem_lock(semid);
          current->semsleeping = NULL;

          if (q->prev) {
               if(sma==NULL)
```

```
                BUG();
            remove_from_queue(q->sma,q);
        }
        if(sma!=NULL)
            sem_unlock(semid);
    }

    for (up = &current->semundo; (u = *up); *up = u->proc_next, kfree(u)) {
        int semid = u->semid;
        if(semid == -1)
            continue;
        sma = sem_lock(semid);
        if (sma == NULL)
            continue;

        if (u->semid == -1)
            goto next_entry;

        if (sem_checkid(sma,u->semid))
            goto next_entry;

        /* remove u from the sma->undo list */
        for (unp = &sma->undo; (un = *unp); unp = &un->id_next) {
            if (u == un)
                goto found;
        }
        printk ("sem_exit undo list error id=%d\n", u->semid);
        goto next_entry;
found:
        *unp = un->id_next;
        /* perform adjustments registered in u */
        nsems = sma->sem_nsems;
        for (i = 0; i < nsems; i++) {
            struct sem * sem = &sma->sem_base[i];
            sem->semval += u->semadj[i];
            if (sem->semval < 0)
                sem->semval = 0; /* shouldn't happen */
            sem->sempid = current->pid;
        }
        sma->sem_otime = CURRENT_TIME;
        /* maybe some queued-up processes were waiting for this */
        update_queue(sma);
next_entry:
        sem_unlock(semid);
    }
    current->semundo = NULL;
}
```

如果当前过程正在（睡眠）等待进入某个临界区，则其 task_struct 结构中的指针 semsleeping 指向

PDF-XCHANGE Click to buy NOW! www.docu-track.com

所在的队列。显然，现在不需要再等待了，所以把当前过程从这个队列中脱链。接着是一个 for 循环，料理那些正在由当前过程所创建的用户空间信号量（即临界区）上操作的过程，告诉它们：信号量已经撤销，临界区已经要“清场”并“关门大吉”，大家请回吧。建议读者在学习了“进程间通信”的有关内容后再回过来自己读一下这段代码。

再看__exit_mm()的代码（kernel/exit.c）：

==================== kernel/exit.c 297 316 ====================

[sys_exit()>do_exit()>__exit_mm()]

```
/*
 * Turn us into a lazy TLB process if we
 * aren't already..
 */
static inline void __exit_mm(struct task_struct * tsk)
{
	struct mm_struct * mm = tsk->mm;

	mm_release();
	if (mm) {
		atomic_inc(&mm->mm_count);
		if (mm != tsk->active_mm) BUG();
		/* more a memory barrier than a real lock */
		task_lock(tsk);
		tsk->mm = NULL;
		task_unlock(tsk);
		enter_lazy_tlb(mm, current, smp_processor_id());
		mmput(mm);
	}
}
```

实际的存储空间释放是调用 mmput()完成的（代码在 fork.c 中），我们已在前一节中读过它的代码，这里要提醒读者的是这里对 mm_relaese()的调用。在 fork()和 execve()两节中，读者已经看到，当 do_fork()时标志位 CLONE_VFORK 为 1 时，父进程在睡眠，等待子进程在一个信号量上执行一次 up()操作以后才能回到用户空间运行，而子进程必须在释放其用户存储空间时执行这个操作，所以这里要通过 mm_release()，在这个信号量上执行一次 up()操作唤醒睡眠中的父进程。其代码已经在 execve()一节中列出，这里不再重复。

将一个进程的 task_struct 结构中的指针 mm 清成 0，这个进程便不再有用户空间了。

回到 do_exit()的代码中，其它几个用于释放资源的函数读者可自行阅读。对于 i386 处理器 exit_thread()是个空函数。

接着，当前进程的状态就改成了 TASK_ZOMBIE，表示进程的生命已经结束，从此不再接受调度。但是当前进程的残骸仍旧占用着最低限度的资源，包括其 task_struct 数据结构和系统空间堆栈所在的两个页面。什么时候释放这两个页面呢？当前进程自己并不释放这两个页面，就像人们自己并不在临终前注销自己的户口一样，而是调用 exit_notify()通知其父进程，让父进程料理后事。

为什么要这样安排，而不是让当前进程，也就是子进程自己照料一切呢？有两个原因。首先，在子进程的 task_struct 数据结构中还有不少有用的统计信息，让父进程来料理后事可以将这些统计信息

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

并入父进程的统计信息中而不会使这些信息丢失。其次，也许更重要的是，系统一旦进入多进程状态以后，任何一刻都需要有个“当前进程”存在。读者在第 3 章中看到了，在中断服务程序以及异常处理程序中都要用到当前进程的系统空间堆栈。如果子进程在系统调度另一个进程投入运行之前就把它的 task_struct 结构和系统空间堆栈释放，那就会造成一个空隙，如果恰好有一次中断或者异常在此空隙中发生就会造成问题。诚然，中断是可以关闭的，可是异常却不能通过关中断来防止其发生，更何况还有“不可屏蔽中断”哩。所以，子进程的 task_struct 结构和系统空间堆栈必须要保存到另一个进程开始运行之后才能释放。这样，让父进程料理后事就是一个合理的安排了。此外，这样安排也有利于使程序简化，否则的话调度程序 schedule()就得要多考虑一些特殊情况了。让我们来看看 exit.c 中函数 exit_notify()的源代码：

==================== kernel/exit.c 323 331 ====================

[sys_exit()>do_exit()>exit_notify()]

```
323  /*
324   * Send signals to all our closest relatives so that they know
325   * to properly mourn us..
326   */
327  static void exit_notify(void)
328  {
329       struct task_struct * p, *t;
330
331       forget_original_parent(current);
```

就像人一样，所谓父进程也有“生父”和“养父”之分。在 task_struct 结构中有个指针 p_opptr 指向其“ original parent”也即生父，另外还有个指针 p_pptr 则指向养父。一个进程在创建之初其生父和养父是一致的，所以两个指针指向同一个父进程。但是，在运行中 p_pptr 可以暂时地改变。这种改变发生在一个进程通过系统调用 ptrace()来跟踪另一个进程的时候，这时候被跟踪进程的 p_pptr 指针被设置成指向正在跟踪它的进程，那个进程就暂时成了被跟踪进程的“养父”。而被跟踪进程的 p_opptr 指针却不变，仍旧指向其生父。如果一个进程在其子进程之前“去世”的话，就要把它的子进程托付给某个进程。托付给谁呢？如果当前进程是一个线程，那就托付给同一线程组中的下一个线程，使子进程的 p_opptr 指向这个线程。否则，就只好托付给系统中的 init 进程，所以这 init 进程就好像是孤儿院。由此可见，所谓“ original parent”也不是永远不变的，原因在于系统中的进程号 pid 以及用作 task_struct 结构的页面都是在周转使用的，所以实际上一来并没有保留这个记录的意义，二来技术上也有困难。现在，当前进程要 exit()了，所以要将其所有的子进程都送进“孤儿院”，要不然到它们也要 exit()的时候就没有父进程来料理它们的后事了。这就是 331 行调用 forget_original_parent()的目的（kernel/exit.c）。

==================== kernel/exit.c 147 174 ====================

[sys_exit()>do_exit()>exit_notify()>forget_original_parent()]

```
147  /*
148   * When we die, we re-parent all our children.
149   * Try to give them to another thread in our process
150   * group, and if no such member exists, give it to
151   * the global child reaper process (ie "init")
152   */
153  static inline void forget_original_parent(struct task_struct * father)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        struct task_struct * p, *reaper;

        read_lock(&tasklist_lock);

        /* Next in our thread group */
        reaper = next_thread(father);
        if (reaper == father)
                reaper = child_reaper;

        for_each_task(p) {
                if (p->p_opptr == father) {
                        /* We dont want people slaying init */
                        p->exit_signal = SIGCHLD;
                        p->self_exec_id++;
                        p->p_opptr = reaper;
                        if (p->pdeath_signal) send_sig(p->pdeath_signal, p, 0);
                }
        }
        read_unlock(&tasklist_lock);
}
```

这段程序中的 for_each_task 在 include/linux/sched.h 中定义为：

==================== include/linux/sched.h 824 825 ====================

```
#define for_each_task(p) \
        for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

就是说，搜索所有的 task_struct 数据结构，凡发现“生父”为当前进程者就将其 p_opptr 指针改成指向 child_reaper，即 init 进程，并嘱其将来 exit()时要发一个 SIGCHLD 信号给 child_reaper，并根据当前进程的 task_struet 结构中的 pdeath_signal 的设置向其发一个信号，告知生父的“噩耗”。

回到 exit_notify()中，下面就来处理由指针 p_pptr 所指向的“养父”进程了。这个父进程就好像是当前进程的“法定监护人”，扮演着更为重要的角色（kernel/exit.c）：

==================== kernel/exit.c 332 419 ====================

[sys_exit()>do_exit()>exit_notify()]

```
        /*
         * Check to see if any process groups have become orphaned
         * as a result of our exiting, and if they have any stopped
         * jobs, send them a SIGHUP and then a SIGCONT.  (POSIX 3.2.2.2)
         *
         * Case i: Our father is in a different pgrp than we are
         * and we were the only connection outside, so our pgrp
         * is about to become orphaned.
         */

        t = current->p_pptr;

        if ((t->pgrp != current->pgrp) &&
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            (t->session == current->session) &&
            will_become_orphaned_pgrp(current->pgrp, current) &&
            has_stopped_jobs(current->pgrp)) {
             kill_pg(current->pgrp,SIGHUP,1);
             kill_pg(current->pgrp,SIGCONT,1);
        }

        /* Let father know we died
         *
         * Thread signals are configurable, but you aren't going to use
         * that to send signals to arbitary processes.
         * That stops right now.
         *
         * If the parent exec id doesn't match the exec id we saved
         * when we started then we know the parent has changed security
         * domain.
         *
         * If our self_exec id doesn't match our parent_exec_id then
         * we have changed execution domain as these two values started
         * the same after a fork.
         *
         */

        if(current->exit_signal != SIGCHLD &&
            ( current->parent_exec_id != t->self_exec_id  ||
              current->self_exec_id != current->parent_exec_id)
            && !capable(CAP_KILL))
             current->exit_signal = SIGCHLD;


        /*
         * This loop does two things:
         *
         * A.  Make init inherit all the child processes
         * B.  Check to see if any process groups have become orphaned
         *   as a result of our exiting, and if they have any stopped
         *   jobs, send them a SIGHUP and then a SIGCONT.  (POSIX 3.2.2.2)
         */

        write_lock_irq(&tasklist_lock);
        current->state = TASK_ZOMBIE;
        do_notify_parent(current, current->exit_signal);
        while (current->p_cptr != NULL) {
             p = current->p_cptr;
             current->p_cptr = p->p_osptr;
             p->p_ysptr = NULL;
             p->ptrace = 0;

             p->p_pptr = p->p_opptr;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                p->p_osptr = p->p_pptr->p_cptr;
                if (p->p_osptr)
                    p->p_osptr->p_ysptr = p;
                p->p_pptr->p_cptr = p;
                if (p->state == TASK_ZOMBIE)
                    do_notify_parent(p, p->exit_signal);
                /*
                 * process group orphan check
                 * Case ii: Our child is in a different pgrp
                 * than we are, and it was the only connection
                 * outside, so the child pgrp is now orphaned.
                 */
                if ((p->pgrp != current->pgrp) &&
                    (p->session == current->session)) {
                    int pgrp = p->pgrp;

                    write_unlock_irq(&tasklist_lock);
                    if (is_orphaned_pgrp(pgrp) && has_stopped_jobs(pgrp)) {
                        kill_pg(pgrp,SIGHUP,1);
                        kill_pg(pgrp,SIGCONT,1);
                    }
                    write_lock_irq(&tasklist_lock);
                }
            }
            write_unlock_irq(&tasklist_lock);
        }
```

代码作者在程序中加了不少注解，代码本身也并不复杂，所以我们基本上把它留给读者自己阅读，不过要给予一些提示。

一个用户 login 到系统中以后，可能会启动许多不同的进程，所有这些进程都使用同一个控制终端（或用来模拟一个终端的窗口）。这些使用同一个控制终端的进程属于同一个 session。此外，用户可以在同一条 shell 命令或执行程序中启动多个进程，例如在命令“ ls | wc -l”中就同时启动了两个进程，这些进程形成一个“组 ” (session 与组是两个不同的概念)。每个 session 或进程组中都有一个为主的、最早创建的进程，这个进程的 pid 就成为 session 和进程组的代号。如果当前进程与父进程属于不同的 session，不同的组，同时又是其所在的组与其父进程之间惟一的纽带，那么一旦当前进程不存在以后，这整个组就成了“孤儿”。在这样的情况下，按 POSIX 3.2.2.2 的规定要给这个进程组中所有的进程都先发一个 SIGHUP 信号，然后再发一个 SIGCONT 信号，这是由 kill_pg()完成的。

我们讲过，exit_notify()最主要的目的是要给父进程发一个信号，让其知道子进程的生命已经结束而来料理子进程的后事，这是通过 do_notify_parent()来完成的。其代码在 kernel/signal.c 中，程序很简单，读者可自行阅读：

==================== kernel/signal.c 732 777 ====================

[sys_exit()>do_exit()>exit_notify()>do_notify_parent()]

```
/*
 * Let a parent know about a status change of a child.
 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

void do_notify_parent(struct task_struct *tsk, int sig)
{
	struct siginfo info;
	int why, status;

	info.si_signo = sig;
	info.si_errno = 0;
	info.si_pid = tsk->pid;
	info.si_uid = tsk->uid;

	/* FIXME: find out whether or not this is supposed to be c*time. */
	info.si_utime = tsk->times.tms_utime;
	info.si_stime = tsk->times.tms_stime;

	status = tsk->exit_code & 0x7f;
	why = SI_KERNEL;	/* shouldn't happen */
	switch (tsk->state) {
	case TASK_STOPPED:
		/* FIXME -- can we deduce CLD_TRAPPED or CLD_CONTINUED? */
		if (tsk->ptrace & PT_PTRACED)
			why = CLD_TRAPPED;
		else
			why = CLD_STOPPED;
		break;

	default:
		if (tsk->exit_code & 0x80)
			why = CLD_DUMPED;
		else if (tsk->exit_code & 0x7f)
			why = CLD_KILLED;
		else {
			why = CLD_EXITED;
			status = tsk->exit_code >> 8;
		}
		break;
	}
	info.si_code = why;
	info.si_status = status;

	send_sig_info(sig, &info, tsk->p_pptr);
	wake_up_parent(tsk->p_pptr);
}
```

参数 tsk 指向当前进程的 task_struct 结构，只有当进程处于 TASK_ZOMBIE（正在 exit()）或 TASK_STOPPED（被跟踪）时才允许调用 do_notify_parent()。从代码中可见，这里的所谓 parent 是指当前进程的“养父”而不是“生父”，也就是由指针 p_pptr 所指而不是 p_opptr 所指的进程。在前面的 forget_original_parent()中已经把每个子进程的 p_opptr 改成了指向 child_reaper，而 notify_parent()中却

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

是向 p_pptr 所指进程发信号；那样，将来当那些子进程要 exit()时岂不是要向一个已经不存在了的父进程发信号吗？不要紧，exit_notify()的代码中随后（392 行）就把子进程的 p_pptr 设置成与 p_opptr 相同。

进程之间都通过亲缘关系连接在一起而形成"关系网"，所用的指针除 p_opptr 和 p_pptr 外，还有：

p_cptr，指向子进程，这里的 c 表示" child"。p_cptr 与 p_pptr 是相对应的。当一个进程有多个子进程时，p_cptr 指向其"最年轻的"，也就是最近创建的那个子进程。

p_ysptr，指向当前进程的"弟弟"，这里的 y 表示" younger"，而 s 表示" sibling"。

p_osptr，指向当前进程的"哥哥"，这里的 o 表示" older"。

这样，当前进程的所有子进程都通过 p_ysptr 和 p_osptr 连接在一起形成一个双链队列。队列中每一个进程的 p_pptr 都指向当前进程，而当前进程的 p_optr 则指向队列中最后创建的子进程。有趣的是，子进程在行事时只认其"养父"，而 p_opptr 所指的"生父"倒似乎无关紧要。当然，一个进程除身处这个由亲属关系形成的队列中之外，同时也身处其它的队列中，所以 task_struct 结构中还有其它的 task_struct 指针，从而形成一个并不简单的"关系网"。进程是在创建的时候在 do_fork()中通过 SET_LINK 进入这个关系网的。SET_LINK 的定义在 include/linux/sched.h 中：

==================== include/linux/sched.h 813 822 ====================

```
#define SET_LINKS(p) do { \
	(p)->next_task = &init_task; \
	(p)->prev_task = init_task.prev_task; \
	init_task.prev_task->next_task = (p); \
	init_task.prev_task = (p); \
	(p)->p_ysptr = NULL; \
	if (((p)->p_osptr = (p)->p_pptr->p_cptr) != NULL) \
		(p)->p_osptr->p_ysptr = p; \
	(p)->p_pptr->p_cptr = p; \
	} while (0)
```

现在，是退出这个关系网的时候了。当 CPU 从 do_notify_parent()返回到 exit_notify()中时，所有子进程的 p_opptr 都已指向 child_reaper，而 p_pptr 仍指向当前进程。随后的 while 循环将子进程队列中的每个进程都转移到 child_reaper 的子进程队列中去，并使其 p_pptr 也指向 child_reaper。同时，对每个子进程也要检查其所属的进程组是否成为了"孤岛"。

如果当前进程是一个 session 中的主进程（current->leader 非 0），那就还要将整个 session 与其主控终端的联系切断，并将该 tty 释放（注意，进程的 task_struct 结构中有个指针 tty 指向其主控终端）。函数 disassociate_ctty()的代码在 drivers/char/tty_io.c 中：

==================== drivers/char/tty_io.c 560 606 ====================

[sys_exit()>do_exit()>exit_notify()>disassociate_ctty()]

```
/*
 * This function is typically called only by the session leader, when
 * it wants to disassociate itself from its controlling tty.
 *
 * It performs the following functions:
 *   (1)  Sends a SIGHUP and SIGCONT to the foreground process group
 *   (2)  Clears the tty from being controlling the session
 *   (3)  Clears the controlling tty for all processes in the
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *       session group.
 *
 * The argument on_exit is set to 1 if called when a process is
 * exiting; it is 0 if called by the ioctl TIOCNOTTY.
 */
void disassociate_ctty(int on_exit)
{
     struct tty_struct *tty = current->tty;
     struct task_struct *p;
     int tty_pgrp = -1;

     if (tty) {
          tty_pgrp = tty->pgrp;
          if (on_exit && tty->driver.type != TTY_DRIVER_TYPE_PTY)
               tty_vhangup(tty);
     } else {
          if (current->tty_old_pgrp) {
               kill_pg(current->tty_old_pgrp, SIGHUP, on_exit);
               kill_pg(current->tty_old_pgrp, SIGCONT, on_exit);
          }
          return;
     }
     if (tty_pgrp > 0) {
          kill_pg(tty_pgrp, SIGHUP, on_exit);
          if (!on_exit)
               kill_pg(tty_pgrp, SIGCONT, on_exit);
     }

     current->tty_old_pgrp = 0;
     tty->session = 0;
     tty->pgrp = -1;

     read_lock(&tasklist_lock);
     for_each_task(p)
          if (p->session == current->session)
               p->tty = NULL;
     read_unlock(&tasklist_lock);
}
```

那么，进程与主控终端的这种联系最初是怎样，以及在什么时候建立的呢？显然，在创建子进程时，将父进程的 task_struct 结构复制给子进程的过程中把结构中的 tty 指针也复制了下来，所以子进程具有与父进程相同的主控终端。但是子进程可以通过 ioctl()系统调用来改变主控终端，也可以先将当前的主控终端关闭然后再打开另一个 tty。不过，在此之前先得通过 setsid()系统调用来建立一个新的人机交互分组（session），并使得作此调用的进程成为该 session 的主进程（leader）。一个 session 的主进程与其主控终端断绝关系意味着整个 session 中的进程都与之断绝了关系，所以要给同一 session 中的进程发出信号。从此以后，这些进程就没有主控终端，成了“后台进程”。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

再回到 do_exit()的代码中。当 CPU 完成了 exit_notify()，回到 do_exit()中时，剩下的大事只有一件了，那就是 schedule()，即进程调度。前面讲过，do_exit()是不返回的，实际上使 do_exit()不返回的正是这里的 schedule()。换言之，在这里对 schedule()的调用是不返回的。当然，在正常条件下对 schedule()的调用是返回的，只不过返回的时机要延迟到本进程再次被调度而进入运行的时候。函数 schedule()按照一定的准则从系统中挑选一个最适合的进程进入运行。这个进程有可能就是正在运行的进程本身，也可能是另一个进程。如果不同的话，那就要进行切换。而当前进程虽然被暂时剥夺了运行权，却维持其“运行状态”，即 task->state 不变，等待下一次又在 schedule()中（由另一个进程引起，或者因中断进入内核后从系统空间返回用户空间之前）被选中时再继续运行，从而从 schedule()中返回。所以，什么时候从 schedule()返回取决于什么时候被进程调度选中而得以继续运行。可是，在这里，当前进程的 task->state 已经变成了 TASK_ZOMBIE，这个条件使它在 schedule()中永远不会再被选中，所以就“黄鹤一去不复返了”。而这里对 schedule()的调用，实际上（从 CPU 的角度看）也是返回的，只不过是返回到另一个进程中去了，只是从当前进程的角度来看没有返回而已。不过，至此为止，当前进程还只是因为不会被选中而不能返回，从理论上说只是无限推迟而已，其 task_struct 结构还是存在的。到父进程收到子进程发来的信号而来料理后事，将子进程的 task_struct 结构释放之时，子进程就最终从系统中消失了。在我们这个情景中，父进程正在 wait4()中等着哩。

像其它系统调用一样，wait4()在内核中的入口是 sys_wait4()，见 kernel/exit.c 中的代码：

==================== kernel/exit.c 487 496 ====================

```
asmlinkage long sys_wait4(pid_t pid,unsigned int * stat_addr, int options, struct rusage * ru)
{
	int flag, retval;
	DECLARE_WAITQUEUE(wait, current);
	struct task_struct *tsk;

	if (options & ~(WNOHANG|WUNTRACED|__WNOTHREAD|__WCLONE|__WALL))
		return -EINVAL;

	add_wait_queue(&current->wait_chldexit,&wait);
```

参数 pid 为某一个子进程的进程号。

首先，在当前进程的系统空间堆栈中通过 DECLARE_WAITQUEUE 分配空间并建立了一个 wait_queue_t 数据结构。有关的宏定义和数据结构都是在 include/linux/wait.h 上中定义的：

==================== include/linux/wait.h 46 56 ====================

```
struct __wait_queue {
	unsigned int flags;
#define WQ_FLAG_EXCLUSIVE	0x01
	struct task_struct * task;
	struct list_head task_list;
#if WAITQUEUE_DEBUG
	long __magic;
	long __waker;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#endif
};
typedef struct __wait_queue wait_queue_t;
```

==================== include/linux/wait.h 92 115 ====================

```
struct __wait_queue_head {
     wq_lock_t lock;
     struct list_head task_list;
#if WAITQUEUE_DEBUG
     long __magic;
     long __creator;
#endif
};
typedef struct __wait_queue_head wait_queue_head_t;

#if WAITQUEUE_DEBUG
# define __WAITQUEUE_DEBUG_INIT(name) \
          , (long)&(name).__magic, 0
# define __WAITQUEUE_HEAD_DEBUG_INIT(name) \
          , (long)&(name).__magic, (long)&(name).__magic
#else
# define __WAITQUEUE_DEBUG_INIT(name)
# define __WAITQUEUE_HEAD_DEBUG_INIT(name)
#endif

#define __WAITQUEUE_INITIALIZER(name,task) \
     { 0x0, task, { NULL, NULL } __WAITQUEUE_DEBUG_INIT(name)}
#define DECLARE_WAITQUEUE(name,task) \
     wait_queue_t name = __WAITQUEUE_INITIALIZER(name,task)
```

也就是说，sys_wait4()一开头就在当前进程的系统堆栈上分配一个 wait_queue_t 数据结构（名为 wait），结构中的 compiler_warning 为 0x1234567，指针 task 指向当前进程的 task_struct，而 list_head 结构 task_list 中的两个指针均为 NULL。由于这个数据结构建立在当前进程的系统空间堆栈中，一旦从 sys_wait4()返回，这个数据结构就不复存在了。与此相应，在进程的 task_struct 中有个 wait_queue_head_t 数据结构 wait_chldexit 用于这个目的。

然后，通过 add_wait_queue()将这个数据结构（wait）加入到当前进程的 wait_chldexit 队列中。这样做的作用在下面重温了 do_notify_parent()的代码以后就会清楚。接着，就进入了一个循环，这是一个不小的循环（kernel/exit.c：sys_wait4()）：

==================== kernel/exit.c 497 583 ====================

[sys_wait4()]

```
repeat:
     flag = 0;
     current->state = TASK_INTERRUPTIBLE;
     read_lock(&tasklist_lock);
     tsk = current;
     do {
          struct task_struct *p;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    for (p = tsk->p_cptr ; p ; p = p->p_osptr) {
        if (pid>0) {
            if (p->pid != pid)
                continue;
        } else if (!pid) {
            if (p->pgrp != current->pgrp)
                continue;
        } else if (pid != -1) {
            if (p->pgrp != -pid)
                continue;
        }
        /* Wait for all children (clone and not) if __WALL is set;
         * otherwise, wait for clone children *only* if __WCLONE is
         * set; otherwise, wait for non-clone children *only*.  (Note:
         * A "clone" child here is one that reports to its parent
         * using a signal other than SIGCHLD.) */
        if (((p->exit_signal != SIGCHLD) ^ ((options & __WCLONE) != 0))
           && !(options & __WALL))
            continue;
        flag = 1;
        switch (p->state) {
        case TASK_STOPPED:
            if (!p->exit_code)
                continue;
            if (!(options & WUNTRACED) && !(p->ptrace & PT_PTRACED))
                continue;
            read_unlock(&tasklist_lock);
            retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
            if (!retval && stat_addr)
                retval = put_user((p->exit_code << 8) | 0x7f, stat_addr);
            if (!retval) {
                p->exit_code = 0;
                retval = p->pid;
            }
            goto end_wait4;
        case TASK_ZOMBIE:
            current->times.tms_cutime += p->times.tms_utime + p->times.tms_cutime;
            current->times.tms_cstime += p->times.tms_stime + p->times.tms_cstime;
            read_unlock(&tasklist_lock);
            retval = ru ? getrusage(p, RUSAGE_BOTH, ru) : 0;
            if (!retval && stat_addr)
                retval = put_user(p->exit_code, stat_addr);
            if (retval)
                goto end_wait4;
            retval = p->pid;
            if (p->p_opptr != p->p_pptr) {
                write_lock_irq(&tasklist_lock);
                REMOVE_LINKS(p);
                p->p_pptr = p->p_opptr;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
553                         SET_LINKS(p);
554                         do_notify_parent(p, SIGCHLD);
555                         write_unlock_irq(&tasklist_lock);
556                     } else
557                         release_task(p);
558                     goto end_wait4;
559                 default:
560                     continue;
561                 }
562             }
563             if (options & __WNOTHREAD)
564                 break;
565             tsk = next_thread(tsk);
566         } while (tsk != current);
567         read_unlock(&tasklist_lock);
568         if (flag) {
569             retval = 0;
570             if (options & WNOHANG)
571                 goto end_wait4;
572             retval = -ERESTARTSYS;
573             if (signal_pending(current))
574                 goto end_wait4;
575             schedule();
576             goto repeat;
577         }
578         retval = -ECHILD;
579     end_wait4:
580         current->state = TASK_RUNNING;
581         remove_wait_queue(&current->wait_chldexit,&wait);
582         return retval;
583     }
```

这个由 goto 实现的循环要到当前进程被调度运行，并且下列条件之一得到满足时才结束（见代码中的“ goto end_wait4”语句）：

- 所等待的子进程的状态变成 TASK_STOPPED 或 TASK_ZOMBIE；
- 所等待的子进程存在，可是不在上述两个状态，而调用参数 options 中的 WNOHANG 标志位为 1，或者当前进程收到了其它的信号；
- 进程号为 pid 的那个进程根本不存在，或者不是当前进程的子进程。

否则，当前进程将其自身的状态设成 TASK_INTERRUPTIBLE（见 499 行）并在 575 行调用 schedule()进入睡眠让别的进程先运行。当该进程因收到信号而被唤醒，并且受到调度从 schedule()返回时，就又经由 576 行的 goto 语句转回 repeat，再次通过一个 for 循环扫描其子进程队列，看看所等待的子进程的状态是否满足条件。这里的 for 循环扫描一个进程的所有子进程，从最年轻的子进程开始沿着由各个 task_struct 结构中的指针 p_osptr 所形成的链扫描，找寻与所等待对象的 pid 相符的子进程、或符合其它一些条件的子进程。这个 for 循环又嵌套在一个 do-while 循环中。为什么要有这个外层的 do-while 循环呢？这是因为当前进程可能是一个线程，而所等待的对象实际上是由同一个进程克隆出来的另一

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

个线程的子进程，所以要通过这个 do-while 循环来检查同一个 thread_group 中所有线程的子进程。代码中的 next_thread()从同一个 thread_group 队列中找到下一个线程的 task_struct 结构，并使局部量 tsk 指向这个结构。在我们这个情景中，当父进程调用 wait4()而第一次扫描其子进程队列时，该子进程尚在运行，所以通过 schedule()进入睡眠。当子进程 exit()时，会向父进程发一个信号，从而将其唤醒。怎么唤醒呢？我们在前面看到，子进程在 exit_notify()中通过 do_notify_parent()向父进程发送信号。这个函数准备下一个 siginfo 数据结构，然后调用 send_sig_info()将其发送给父进程，并调用 wake_up_process()将父进程唤醒。对 send_sig_info()的代码我们将在“进程间通信”的信号一节中介绍。而 wake_up_process()，则把父进程的状态从 TASK_INTERRUPTIBLE 改成 TASK_RUNNING，并将其转移到可执行队列中，使 schedule()能够“看”到父进程而可以调度其运行。

当父进程因子进程在 exit()向其发送信号而被唤醒时，就转回到前面 sys_wait4()中的 repeat 处，又一次扫描其子进程队列。这一次，子进程的状态已经改成 TASK_ZOMBIE 了，所以父进程在将子进程在用户空间运行的时间和系统空间运行的时间两项统计数据合并入其自身的统计数据中。然后，在典型的条件下，就调用 release_task()将子进程残存的资源，就是其 task_sturct 结构和系统空间堆栈，全都释放（kernel/exit.c）：

==================== kernel/exit.c 25 68 ====================

[sys_wait4()>release()]:

```
static void release_task(struct task_struct * p)
{
    if (p != current) {
#ifdef CONFIG_SMP
        /*
         * Wait to make sure the process isn't on the
         * runqueue (active on some other CPU still)
         */
        for (;;) {
            task_lock(p);
            if (!p->has_cpu)
                break;
            task_unlock(p);
            do {
                barrier();
            } while (p->has_cpu);
        }
        task_unlock(p);
#endif
        atomic_dec(&p->user->processes);
        free_uid(p->user);
        unhash_process(p);

        release_thread(p);
        current->cmin_flt += p->min_flt + p->cmin_flt;
        current->cmaj_flt += p->maj_flt + p->cmaj_flt;
        current->cnswap += p->nswap + p->cnswap;
        /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
             * Potentially available timeslices are retrieved
             * here - this way the parent does not get penalized
             * for creating too many processes.
             *
             * (this cannot be used to artificially 'generate'
             * timeslices, because any timeslice recovered here
             * was given away by the parent in the first place.)
             */
            current->counter += p->counter;
            if (current->counter >= MAX_COUNTER)
                current->counter = MAX_COUNTER;
            free_task_struct(p);
        } else {
            printk("task releasing itself\n");
        }
    }
```

这里通过 unhash_process()把子进程的 task_struct 结构从杂凑表队列中摘除，然后把子进程的其它几项统计信息也合并入父进程。至于 release_thread()只是检查进程的 LDT（如果有的话）是否确已释放。最后，就调用 free_task_struct()将 task_struct 结构和系统空间堆栈所占据的两个物理页面释放。

在 sys_wait4()中还有个特殊情况需要考虑，那就是万一子进程的 p_opptr 与 p_pptr 不同，也就是说其“养父”与“生父”不同。如前所述，进程在 exit()时，do_notify_parent()的对象是其“养父”，但是当“生父”与“养父”不同时，其“生父”可能也在等待，所以将子进程的 p_pptr 指针设置成与 p_opptr 相同，并通过 REMOVE_LINKS 将其 task_struct 从其“养父”的队列中脱离出来，再通过 SET_LINKS 把它归还给“生父”，重新挂入其“生父”的队列。然后，给其“生父”发一信号，让它自己来处理。

此外，根据当前进程在调用 wait4()时的要求，还可能要把一些状态信息和统计信息通过 put_user()复制到用户空间中。如果复制失败的话，那暂时就不能将子进程的 task_struct 结构释放了（这里的“ goto end_wait4”跳过了对 release()的调用）。在这种情况下，系统中会留下子进程的“尸体”，用户通过“ ps ”命令来观察系统中的进程状态时，会看到有个进程的状态为“ ZOMBIE”。读者在前面看到：在 exit_notify()中，当父进程要结束生命前为其子进程“托孤”时，还要看一下子进程的状态是否 TASK_ZOMBIE，若是的话，就要替它调用 do_notify_parent()给新的“养父”发一信息，就是这个原因。

至此，在执行了 release()以后，子进程就最终“灰飞烟灭”，从系统中消失了。

可是，要是父进程不在 wait4()中等待呢？那也不要紧。读者在第 3 章中已经看到，每当进程从系统调用、中断或异常返回时，都要检查一下是否有信号等待处理，如有的话就转入 entry.S 中的 signal_return 处调用 do_signal()。而 do_signal()中有一个片段为（arch/i386/kernel/signal.c 中）：

==================== arch/i386/kernel/signal.c 643 651 ====================

```
            ka = &current->sig->action[signr-1];
            if (ka->sa.sa_handler == SIG_IGN) {
                if (signr != SIGCHLD)
                    continue;
                /* Check for SIGCHLD: it's special.  */
                while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
                    /* nothing */;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                continue;
            }
```

可见父进程在收到 SIGCHLD 信号后还会被动地来调用 sys_wait4()，此时的调用参数 pid 为-1，表示同一个进程组中的任何一个子进程都在处理之列（见 sys_wait4()的 for 循环中对参数 pid 的比对）。当然，如果父进程已经为 SIGCHLD 信号设置了其它的处理程序，那就另作别论了。

读者也许还会问，怎样才能保证一定会有系统调用、中断或异常来迫使其父进程执行 do_signal()呢？万一父进程在运行时既不作系统调用，也不访问外设，更没有任何操作引起异常呢？别忘记时钟中断是周期性地发生的，要不然就连调度也有可能不会发生了，正因为如此，时钟中断才被看作是系统的“心跳”。

4.6 进程的调度与切换

在多进程的操作系统中，进程调度是一个全局性的、关键性的问题，它对系统的总体设计、系统的实现、功能设置以及各方面的性能都有着决定性的影响。根据调度结果所作的进程切换的速度，也是衡量一个操作系统性能的重要指标。进程调度机制的设计，还对系统复杂性有着极大的影响，常常会由于实现的复杂程度而在功能与性能方面作出必要的权衡和让步。一个好的系统的进程调度机制，要兼顾三种不同应用的需要：

- 交互式应用。在这种应用中，着重于系统的响应速度，使共用一个系统的各个用户（以及各个应用程序）都能感觉到自己是在独占地使用一个系统。特别是，当系统中有大量进程共存时，仍要能保证每个用户都有可以接受的响应速度而并不感到明显的延迟。根据测定，当这种延迟超过 150 毫秒时，使用者就会明显地感觉到。
- 批处理应用。批处理应用往往是作为“后台作业”运行的，所以对响应速度并无要求，但是完成一个作业所需的时间仍是一个重要的因素，考虑的是“平均速度”。
- 实时应用。这是时间性最强的应用，不但要考虑进程执行的平均速度，还要考虑“即时速度”；不但要考虑响应速度（即从某个事件发生到系统对此作出反应并开始执行有关程序之间所需的时间），还要考虑有关程序（常常在用户空间）能否在规定时间内执行完。在实时应用中，注重的是对程序执行的“可预测性”。

另外，进程调度的机制还要考虑到“公正性”，让系统中的所有进程都有机会向前推进，尽管其进度各有不同，并最终受到 CPU 速度和负载的影响。更重要的是，还要防止“死锁”的发生，以及防止对 CPU 能力的不合理使用，也就是说要防止 CPU 尚有能力且有进程等着执行，却由于某种原因而长时间得不到执行的情况。一旦这些情况发生时，调度机制还应能识别与化解。可以说，关于进程调度的研究是整个操作系统理论的核心。不过，本书的目的在于对 Linux 内核的剖析和解释，而不在于理论方面的深入探讨，有兴趣的读者可以阅读操作系统方面的专著。

为了满足上述的目标，在设计一个进程调度机制时要考虑的具体问题主要有：

(1) 调度的时机：在什么情况下、什么时候进行调度。

(2) 调度的“政策”（policy）：根据什么准则挑选下一个进入运行的进程。

(3) 调度的方式：是“可剥夺”（preemptive）还是“不可剥夺”（nonpreemptive）。当正在运行的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进程并不自愿暂时放弃对 CPU 的“使用权”时，是否可以强制性地暂时剥夺其使用权，停止其运行而给其它的进程一个机会。如果是可剥夺，那么是否在任何条件下都可剥夺，有没有“例外”？

这三个问题，特别是第一和第三个问题，是紧密结合在一起的。例如，如果调度的性质是绝对地不可剥夺，也就是说坚持完全自愿的原则，那么调度的时机也就基本上决定了，只能在有一个进程自愿调度的时候才能进行调度。相应地，就要设计一个“原语”，即系统调用，让进程可以表达自己的这个意愿。同时，还要考虑，如果一个进程因陷入了死循环而抓住 CPU 不放该怎么办。

这里要说明一下，在中文里也许应该把是否可以剥夺称为“政策”，但是在英文的书刊中已经把调度准则或标准称为“ policy”，所以我们只好把这称为“方式”，以免引起不必要的混淆。

进一步，如果调度的性质是有条件地可剥夺，那么，在什么情况下可剥夺就成了重要的问题。例如，可以把时间划分成时间片，每个时间片来一次时钟中断，而调度可以在时间片中断时进行。按进程的优先级别的高低进行调度，每个时间片一次，除此之外就只能在进程自愿时才可进行调度。这样，只要时间片划分得当，交互式应用的要求就可以满足了。但是，这样的系统显然不适合实时的应用。因为，有可能发生“急惊风遇上慢郎中”的情况，优先级别高的进程急着要运行，而正在运行中的进程偏偏“觉悟不高”，不懂得“先人后己”，别的进程只好干等着它把时间片用完而坐失良机。从另一个角度讲，这也取决于技术的发展，特别是 CPU 的速度。例如，就在这么一个系统中，如果可以把时间片分小到 0.5 毫秒，而 CPU 仍能在这么短的时间里做足够多的事，那么对一般的实时应用来说可能还是能满足要求的，虽然从整体上讲 CPU 用在调度与切换上的开销所占的比例上升了。

那么，Linux 内核的调度机制到底是什么样的呢？我们还是分三个方面来同答这个问题。

在往下叙述之前，此处先给出一个进程的状态转换关系示意图（图 4.4）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

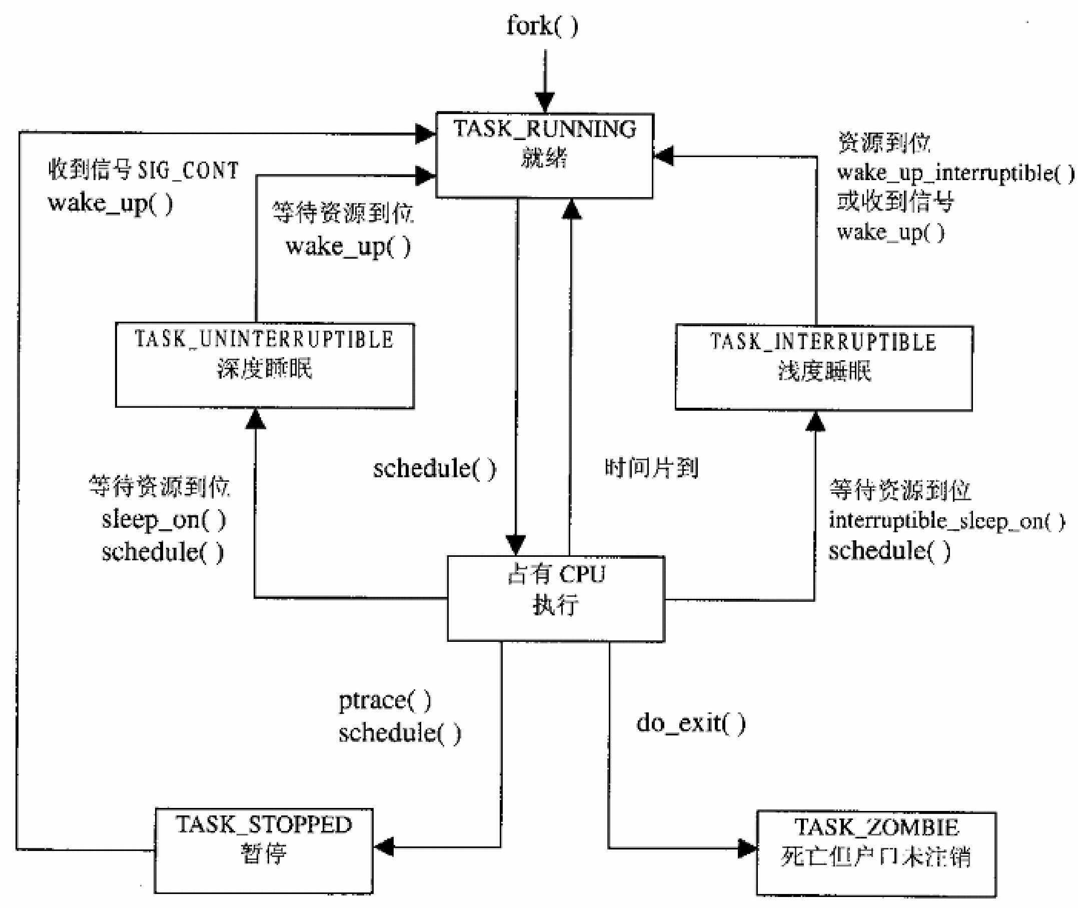


图 4.4　进程状态转换图

先看调度的时机。

首先，自愿的调度随时都可以进行。在内核里面，一个进程可以通过 schedule()启动一次调度，当然也可以在调用 schedule()之前，将本进程的状态设置成为 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE，暂时放弃运行而进入睡眠。在用户空间中，则可以通过系统调用 pause()来达到同样的目的。也可以为这种自愿的暂时放弃运行加上时间限制。在内核中有 schedule_timeout()用于此项目的；相应地，在用户空间则可以通过系统调用 nanosleep()而达到目的（注意，sleep()是库函数，不是系统调用，但最终要通过系统调用来完成）。这里要指出：从应用的角度看，只有在用户空间自愿放弃运行这一举动是可见的；而在内核中自愿放弃运行则是不可见的，它隐藏在其它可能受阻的系统调用中。几乎所有涉及到外设的系统调用，如 open()、read()、write()和 select()等，都是可能受阻的。

除此之外，调度还可以非自愿地，即强制地发生在每次从系统调用返回的前夕，以及每次从中断或异常处理返回到用户空间的前夕。注意，这里“返回到用户空间”几个字是关键性的，因为这意味着只有在用户空间（当 CPU 在用户空间运行时）发生的中断或异常才会引起调度。关于这一点我们在第 3 章讲述中断返回时提到过，但是有必要在此加以强调，并重温 arch/i386/kernel/entry.S 中的两个片

段：

==================== arch/i386/kernel/entry.S 260 261 ====================

```
260 ret_from_exception:
261 #ifdef CONFIG_SMP
```

==================== arch/i386/kernel/entry.S 267 279 ====================

```
267 #else
268      movl SYMBOL_NAME(irq_stat),%ecx      # softirq_active
269      testl SYMBOL_NAME(irq_stat)+4,%ecx   # softirq_mask
270 #endif
271      jne   handle_softirq
272
273 ENTRY(ret_from_intr)
274      GET_CURRENT(%ebx)
275      movl EFLAGS(%esp),%eax      # mix EFLAGS and CS
276      movb CS(%esp),%al
277      testl $(VM_MASK | 3),%eax   # return to VM86 mode or non-supervisor?
278      jne ret_with_reschedule
279      jmp restore_all
```

277 行中寄存器 EAX 的内容有两个来源，其最低的字节来自保存在堆栈中的进入中断前夕段寄存器 CS 的内容，最低的两位表示当时的运行级别。从代码中可以看到，转入 ret_with_reschedule 的条件为中断（或异常）发生前 CPU 的运行级别为 3，即用户状态（我们在这里不关心 VM_MASK，那是为 VM86 模式而设置的）。这一点对于系统的设计和实现有很重要的意义。因为那意味着当 CPU 在内核中运行时无需考虑强制调度的可能性。发生在系统空间的中断或异常当然是可能的，但是这种中断或异常不会引起调度。这就使内核的实现简化了，早期的 Unix 内核正是靠这个前提来简化其设计与实现的。否则的话，内核中所有可能为一个以上进程共享的变量和数据结构就全都要通过互斥机制（如信号量）加以保护，或者说都要放在临界区中。不过，随着多处理器 SMP 系统结构的出现以及日益广泛的采用，这种简化正在失去重要性。在多处理器 SMP 系统中（见“多处理器 SMP 系统结构”一章），尽管在内核中由于不会发生调度而无需考虑互斥，但却不能不考虑在另一个处理器上运行的进程访问共享资源的可能性。这样，不管在同一个 CPU 上是否有可能在内核中发生调度，所有可能为多个进程（可能在不同的 CPU 上运行）共享的变量和数据结构，都得保护起来。这就是读者在阅读代码时看到那么多的 up()、down()等信号量操作或加锁操作的原因。Linux 内核中一般将用于多处理器 SMP 结构的代码放在条件编泽#ifdef __SMP__中，但是却没有把这些用于互斥保护的操作也放在条件编译中。究其原因，一来可能是太多了，加不胜加，再说在单处理器条件下的运行时开销也不大；二来也是为日后对调度机制的改进奠定基础。

那么 Linux 现行的这种调度机制有什么缺点或不足，为什么可能会有日后的改进呢？例如：在实时的应用中，某个中断的发生可能不但要求迅速的中断服务，还要求迅速地调度有关进程进入运行，以便在较高的层次上，也就是在用户空间中对事件进行及时的处理。可是，如果这样的中断发生在内核中时，本次中断返回是不会引起调度的，而要到最初使 CPU 从用户空间进入内核的那次系统调用或中断（或异常）返回时才会发生调度。倘若内核中的这段代码恰好需要较长时间完成的话，或者连续又发生几次中断的话，就可能将调度过分地推迟。良好的内核代码可以减轻这个问题，但并不能从根本上解决问题。所以这是个设计问题而不是实现问题。只是，随着 CPU 速度变得越来越快，这个问题

渐渐地变得不那么重要了。

注意，“从系统空间返回到用户空间”只是发生调度的必要条件，而不是充分条件。具体是否发生调度还要看有无此种要求，看一下 arch/i386/kernel/entry.S 中的这一段代码：

==================== arch/i386/kernel/entry.S 217 223 ====================

```
ret_with_reschedule:
	cmpl $0,need_resched(%ebx)
	jne reschedule
	cmpl $0,sigpending(%ebx)
	jne signal_return
restore_all:
	RESTORE_ALL
```

==================== arch/i386/kernel/entry.S 287 289 ====================

```
reschedule:
	call SYMBOL_NAME(schedule)    # test
	jmp ret_from_sys_call
```

可见，只有在当前进程的 task_struct 结构中的 need_resched 字段为非 0 时才会转到 reschedule 处调用 schedule()。那么，谁来设置这个字段呢？当然是内核，从用户空间是访问不到进程的 task_struct 结构的。可是，内核在什么情况下设置这个字段呢？除当前进程通过系统调用自愿让出运行以及在系统调用中因某种原因受阻以外，主要就是当因某种原因唤醒一个进程的时候，以及在时钟中断服务程序发现当前进程已经连续运行太久的时候。

再看调度的方式。

Linux 内核的调度方式可以说是“有条件的可剥夺”方式。当进程在用户空间运行时，不管自愿不自愿，一旦有必要（例如已经运行了足够长的时间），内核就可以暂时剥夺其运行而调度其它进程进入运行。可是，一旦进程进入了内核空间，或者说进入了“长官”（supervisor）模式，那就好像是进了“高层”而“刑不上大夫”了。这时候，尽管内核知道应该要调度了，但实际上却不会发生，一直要到该进程即将“下台”，也就是回到用户空间的前夕才能剥夺其运行。所以，Linux 的调度方式从原则上来说是可剥夺的，可是在实际运行中由于调度时机的限制而变成了有条件的。正因为这样，有的书说 Linux 的调度是可剥夺的，有的却说是不可剥夺的，甚至同一本书中有时候说是可剥夺的，有时候又说是不可剥夺的，其原因盖出于此。

那么，剥夺式的调度发生在什么时候呢？同样也是发生在进程从系统空间（包括因系统调用进入内核）返回用户空间的前夕。

至于调度政策，基本上是从 UNIX 继承下来的以优先级为基础的调度。内核为系统中的每个进程计算出一个反映其运行“资格”的权值，然后挑选权值最高的进程投入运行。在运行过程中，当前进程的资格随时间而递减，从而在下一次调度的时候原来资格较低的进程可能就更有资格运行了。到所有进程的资格都变成了 0 时，就重新计算一次所有进程的资格。资格的计算主要是以优先级为基础的，所以说是以优先级为基础的调度。

但是，为了适应各种不同应用的需要，内核在此基础上实现了三种不同的政策：SCHED_FIFO、SCHED_RR 以及 SCHED_OTHER。每个进程都有自己适用的调度政策，并且，进程还可以通过系统调用 sched_setscheduler()设定自己适用的调度政策。其中 SCHED_FIFO 适合于时间性要求比较强、但每

次运行所需的时间比较短的进程，实时的应用大都具有这样的特点。SCHED_RR 中的“ RR”表示“ Round Robin”，是轮流的意思，这种政策适合比较大、也就是每次运行需时较长的进程。而除此二者之外的 SCHED_OTHER，则为传统的调度政策，比较适合于交互式的分时应用。

既然每个进程都有自己的适用调度政策，内核怎样在适用不同调度政策的进程之间决定取舍呢？实际上最后还是都归结到各个进程的权值，只不过是在计算资格时把适用政策也考虑进去，就好像考大学时符合某些特殊条件的考生可以获得加分一样。同时，对于适用不同政策的进程的优先级别也加了限制。我们将结合代码更深入地讨论这些政策间的差异和作用。

下面，我们就结合代码深入到调度和切换的过程中去。在本节中我们先看一个主动调度，也就是由当前进程自愿调用 schedule()暂时放弃运行的情景。在 exit()一节中，读者已经看到一个正在结束生命的进程在 do_exit()中的最后一件事就是调用 schedule()，我们就从这里接着往下看，深入到 schedule()里面去，其代码在 kernel/sched.c 中：

==================== kernel/sched.c 498 529 ====================

```
/*
 *  'schedule()' is the scheduler function. It's a very simple and nice
 * scheduler: it's not perfect, but certainly works for most things.
 *
 * The goto is "interesting".
 *
 *   NOTE!!  Task 0 is the 'idle' task, which gets called when no other
 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
 * information in task[0] is never used.
 */
asmlinkage void schedule(void)
{
	struct schedule_data * sched_data;
	struct task_struct *prev, *next, *p;
	struct list_head *tmp;
	int this_cpu, c;

	if (!current->active_mm) BUG();
need_resched_back:
	prev = current;
	this_cpu = prev->processor;

	if (in_interrupt())
		goto scheduling_in_interrupt;

	release_kernel_lock(prev, this_cpu);

	/* Do "administrative" work here while we don't hold any locks */
	if (softirq_active(this_cpu) & softirq_mask(this_cpu))
		goto handle_softirq;
handle_softirq_back:

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这个函数中使用了许多 goto 语句。对于这么一个非常频繁地执行的函数，把运行效率放在第一位是可以理解的，只是给阅读和理解带来了一些困难。

以前我们讲过，在 task_struct 结构中有两个 mm_struct 指针。一个是 mm，指向代表着进程的虚存（用户）空间的数据结构。如果当前进程实际上是个内核线程，那就没有用户空间，所以其 mm 指针为 0，运行时就要暂时借用在它之前运行的那个进程的 active_mm。所以，正在运行中的进程，也即当前进程，在进入 schedule()时其 active_mm 一定不能是 0（见 515 行）。后面我们还要回到这个话题上。

以前讲过，对 schedule()只能由进程在内核中主动调用，或者在当前进程从系统空间返回用户空间的前夕被动地发生，而不能在一个中断服务程序的内部发生。即使一个中断服务程序有调度的要求，也只能通过把当前进程的 need_resched 字段设成 1 来表达这种要求，而不能直接调用 schedule()。读者也许会问，我们在第 3 章中看到，在执行中断服务程序的时候是允许开中断的，如果在执行过程中发生了嵌套中断，那么当从嵌套的中断返回时不是也要调用 schedule()吗？那不就等于是在中断服务程序的内部调用了这个函数吗？其实，从嵌套的中断返回时不会调用 schedule()，因为此时的中断返回并不是返回到用户空间。还要注意：因中断而进入内核并不等于已经进入了某个中断服务程序，而当 CPU 要从系统空间返回用户空间之时则已经离开了具体的中断服务程序，详见第 3 章。所以，如果在某个中断服务程序内部调用 schedule()，那一定是有问题了，所以转向 scheduling_in_interrupt。接着看 kernel/sched.c：

==================== kernel/sched.c 686 689 ====================

[schedule()]

```
scheduling_in_interrupt:
	printk("Scheduling in interrupt\n");
	BUG();
	return;
```

内核对此的反应是显示或者在/var/log/messages 文件末尾添上一条出错信息，然后执行一个宏操作 BUG，这是在 include/asm-i386/page.h 中定义的：

==================== include/asm-i386/page.h 85 92 ====================

```
85  /*
86   * Tell the user there is some problem. Beep too, so we can
87   * see^H^H^Hhear bugs in early bootup as well!
88   */
89  #define BUG() do { \
90  	printk("kernel BUG at %s:%d!\n", __FILE__, __LINE__); \
91  	__asm__ __volatile__(".byte 0x0f,0x0b"); \
92  } while (0)
```

这里的奥妙之处是在 91 行中准备下了两个字节 0x0f 和 0x0b，让 CPU 当作指令去执行。可是由这两个字节构成的是非法指令，因而会产生一次“非法指令（invalid_op）”异常，使 CPU 执行 do_invalid_op()。当然，在实际运行中这样的错误（在中断服务程序或 bf 函数的内部调用 schedule()）是不会发生的，除非正在调试用户自己编写的中断服务程序。

我们回过头来继续往下看 schedule()，这里 523 行的 release_kernel_lock()对于 i386 单处理器系统为空语句，所以接着就是检查是否有内核软中段服务请求在等待（见第 3 章）。如果有就转入 handle_softirq

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

为这些请求服务：

==================== kernel/sched.c 675 677 ====================
[schedule()]

```
handle_softirq:
	do_softirq();
	goto handle_softirq_back;
```

从执行 softirq 队列完毕以后继续往下看：

==================== kernel/sched.c 528 541 ====================
[schedule()]

```
handle_softirq_back:

	/*
	 * 'sched_data' is protected by the fact that we can run
	 * only one process per CPU.
	 */
	sched_data = & aligned_data[this_cpu].schedule_data;

	spin_lock_irq(&runqueue_lock);

	/* move an exhausted RR process to be last.. */
	if (prev->policy == SCHED_RR)
		goto move_rr_last;
move_rr_back:
```

指针 sched_data 指向一个 schedule_data 数据结构，用来保存供下一次调度时使用的信息（kernel/sched.c）：

==================== kernel/sched.c 91 101 ====================

```
/*
 * We align per-CPU scheduling data on cacheline boundaries,
 * to prevent cacheline ping-pong.
 */
static union {
	struct schedule_data {
		struct task_struct * curr;
		cycles_t last_schedule;
	} schedule_data;
	char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { {{&init_task,0}}};
```

这里的类型 cycles_t 实际上是无符号整数，用来记录调度发生的时间。这个数据结构是为多处理器 SMP 结构而设的，所以我们在这里并不关心。数组中的第一个元素，即 CPU0 的 schedule_data 结构初始化成{&init_task, 0}，其余的则全为{0, 0}。代码中__cacheline_aligned 表示数据结构的起点应与高速缓存中的缓冲线对齐。

下面就要涉及可执行进程队列了，所以先将这个队列锁住，以防止来自其它处理器的干扰。如果

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

当前进程 prev 的调度政策为 SCHED_RR，即轮换调度，那就要先进行一点特殊的处理。SCHED_RR 和 SCHED_FIFO 都是基于优先级的调度政策，可是在怎样调度具有相同优先级的进程这个问题上二者有区别。调度政策为 SCHED_FIFO 的进程一旦受到调度而开始运行之后，就要一直运行到自愿让出或者被优先级更高的进程剥夺为止。对于每次受到调度时要求运行时间不长的进程，这样并没有什么不妥。可是，如果是受到调度后可能会长时间运行的进程，那样就不公平了。这种不公正性是对具有相同优先级的进程而言的。因为具有更高优先级的进程可以剥夺它的运行，而优先级更低的进程则本来就没有机会运行。但是，这样对具有相同优先级的其它进程就不公平了。所以，对这样的进程应该实行 SCHED_RR 调度政策，这种政策在相同的优先级上实行轮换调度。也就是说，对调度政策为 SCHED_RR 的进程有个时间配额，用完了这个配额就要让具有相同优先级的其它就绪进程先运行。这里，就是对调度政策为 SCHED_RR 的当前进程的这种处理（kernel/sched.c）：

==================== kernel/sched.c 679 685 ====================

[schedule()]

```
move_rr_last:
	if (!prev->counter) {
		prev->counter = NICE_TO_TICKS(prev->nice);
		move_last_runqueue(prev);
	}
	goto move_rr_back;

```

这是什么意思呢？这里的 prev->counter 代表着当前进程的运行时间配额，其数值在每次时钟中断时都要递减。这是在一个函数 update_process_times()中进行的，详见下一节。不管一个进程的时间配额有多高，随着运行时间的积累最终总会递减到 0。对于调度政策为 SCHED_RR 的进程，一旦其时间配额降到了 0，就要从可执行进程队列 runqueue 中当前的位置上移到队列的末尾，同时恢复其最初的时间配额。对于具有相同优先级的进程，调度的时候排在前面的进程优先，所以这使队列中具有相同优先级的其它进程有了优势。宏操作 NICE_TO_TICKS 根据系统时钟的精度将进程的优先级别换算成可以运行的时间配额，这也是在 kernel/sched.c 中定义的：

==================== kernel/sched.c 44 67 ====================

```
/*
 * Scheduling quanta.
 *
 * NOTE! The unix "nice" value influences how long a process
 * gets. The nice value ranges from -20 to +19, where a -20
 * is a "high-priority" task, and a "+10" is a low-priority
 * task.
 *
 * We want the time-slice to be around 50ms or so, so this
 * calculation depends on the value of HZ.
 */
#if HZ < 200
#define TICK_SCALE(x)	((x) >> 2)
#elif HZ < 400
#define TICK_SCALE(x)	((x) >> 1)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#elif HZ < 800
#define TICK_SCALE(x)  (x)
#elif HZ < 1600
#define TICK_SCALE(x)  ((x) << 1)
#else
#define TICK_SCALE(x)  ((x) << 2)
#endif

#define NICE_TO_TICKS(nice) (TICK_SCALE(20-(nice))+1)
```

将一个进程的 task_struct 结构从可执行队列中的当前位置移到队列的末尾是由 move_last_runqueue 完成的（kernel/sched.c）：

==================== kernel/sched.c 309 313 ====================

[schedule()>move_last_runqueue()]

```
static inline void move_last_runqueue(struct task_struct * p)
{
	list_del(&p->run_list);
	list_add_tail(&p->run_list, &runqueue_head);
}
```

把进程移到可执行进程队列的末尾意味着：如果队列中没有资格更高的进程，但是有一个资格与之相同的进程存在，那么，那个资格虽然相同而排在前面的进程就会被选中。继续在 schedule()中往下看（kernel/sched.c）：

==================== kernel/sched.c 541 553 ====================

[schedule()]

```
move_rr_back:

	switch (prev->state) {
		case TASK_INTERRUPTIBLE:
			if (signal_pending(prev)) {
				prev->state = TASK_RUNNING;
				break;
			}
		default:
			del_from_runqueue(prev);
		case TASK_RUNNING:
	}
	prev->need_resched = 0;
```

当前进程就是正在运行中的进程，可是当进入 schedule()时其状态却不一定是 TASK_RUNNING。例如，在我们这个情景中，当前进程已在 do_exit()中将其状态改成 TASK_ZOMBIE。又例如，前一节中我们看到当前进程在 sys_wait4()中调用 schedule()时的状态为 TASK_INTERRUPTIBLE。所以，这里的 prev->state 与其说是当前进程的状态还不如说是其意愿。正因为这样，当其意愿既不是继续进行也不是可中断的睡眠时，就要通过 del_from_runqueue()把这进程从可执行队列中撤下来。另一方面，也可以看出 TASK_INTERRUPTIBLE 与 TASK_UNINTERRUPTIBLE 两种睡眠状态之间的区别，前者在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进程有信号等待处理时要将其改成 TASK_RUNNING，让其处理完这些信号再说，而后者则不受信号的影响。请注意，在 548 行与 549 行之间并无 break 语句，所以当没有信号等待处理时就落入了 default 的情形，同样要将进程从可执行队列中撤下来。反之，如果当前进程的意愿是 TASK_RUNNING，即继续进行（见 551 行），那在这里就不需要有什么特殊处理。

然后，将 prev->need_resched 恢复成 0，因为所需求的调度已经在进行。下面就要挑选一个进程来运行了（kernel/sched.c）：

==================== kernel/sched.c 555 576 ====================

[schedule()]

```
	/*
	 * this is the scheduler proper:
	 */

repeat_schedule:
	/*
	 * Default process to select..
	 */
	next = idle_task(this_cpu);
	c = -1000;
	if (prev->state == TASK_RUNNING)
		goto still_running;

still_running_back:
	list_for_each(tmp, &runqueue_head) {
		p = list_entry(tmp, struct task_struct, run_list);
		if (can_schedule(p, this_cpu)) {
			int weight = goodness(p, this_cpu, prev->active_mm);
			if (weight > c)
				c = weight, next = p;
		}
	}
```

在这段程序中，next 总是指向已知最佳的候选进程，c 则是这个进程的综合权值，或运行资格。挑选的过程从 idle 进程即 0 号进程开始，其权值为-1000，这是可能的最低值，表示仅在没有其它进程可以运行时才会让它运行。然后，遍历可执行队列 runqueue 中的每个进程（在单 CPU 系统中 can_schedule() 的返回值永远为 1），也就是一般操作系统教科书中所称的“就绪”进程，为每一个这样的进程通过函数 goodness()计算出它当前所具有的权值，然后与当前的最高值 c 相比。注意这里的条件“ weight > c”，这意味着“先入为大”。也就是说，如果两个进程具有相同权值的话，那就是排在前面的进程胜出。代码中的 list_for_each 是个宏定义，定义于 include/linux/list.h 上：

==================== include/linux/list.h 144 150 ====================

```
/**
 * list_for_each	-	iterate over a list
 * @pos:	the &struct list_head to use as a loop counter.
 * @head:	the head for your list.
 */
```

```
#define list_for_each(pos, head) \
    for (pos = (head)->next; pos != (head); pos = pos->next)
```

这里还有一个小插曲，就是如果当前进程的意图是继续运行，那就要先执行一下 still_running（kernel/sched.c）:

==================== kernel/sched.c 670 674 ====================

[schedule()]

```
still_running:
    c = goodness(prev, this_cpu, prev->active_mm);
    next = prev;
    goto still_running_back;

```

也就是说，如果当前进程想要继续运行，那么在挑选候选进程时以当前进程此刻的权值开始。这意味着，相对于权值相同的其它进程来说，当前进程优先。

那么，进程的当前权值是怎样计算的呢？请看 goodness()的代码（kernel/sched.c）:

==================== kernel/sched.c 123 187 ====================

[schedule()>goodness()]

```
/*
 * This is the function that decides how desirable a process is..
 * You can weigh different processes against each other depending
 * on what CPU they've run on lately etc to try to handle cache
 * and TLB miss penalties.
 *
 * Return values:
 *   -1000: never select this
 *       0: out of time, recalculate counters (but it might still be
 *        selected)
 *     +ve: "goodness" value (the larger, the better)
 *   +1000: realtime process, select this.
 */

static inline int goodness(struct task_struct * p, int this_cpu, struct mm_struct *this_mm)
{
    int weight;

    /*
     * select the current process after every other
     * runnable process, but before the idle thread.
     * Also, dont trigger a counter recalculation.
     */
    weight = -1;
    if (p->policy & SCHED_YIELD)
        goto out;

    /*
     * Non-RT process - normal case first.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 */
	if (p->policy == SCHED_OTHER) {
		/*
		 * Give the process a first-approximation goodness value
		 * according to the number of clock-ticks it has left.
		 *
		 * Don't do any other calculations if the time slice is
		 * over..
		 */
		weight = p->counter;
		if (!weight)
			goto out;

#ifdef CONFIG_SMP
		/* Give a largish advantage to the same processor...   */
		/* (this is equivalent to penalizing other processors) */
		if (p->processor == this_cpu)
			weight += PROC_CHANGE_PENALTY;
#endif

		/* .. and a slight advantage to the current MM */
		if (p->mm == this_mm || !p->mm)
			weight += 1;
		weight += 20 - p->nice;
		goto out;
	}

	/*
	 * Realtime process, select the first one on the
	 * runqueue (taking priorities within processes
	 * into account).
	 */
	weight = 1000 + p->rt_priority;
out:
	return weight;
}
```

首先，如果一个进程通过系统调用 sched_yield()明确表示了“礼让”后，就将其权值定为-1。这是很低的权值，一般就绪进程的权值至少是 0。

对于没有实时要求的进程，即调度政策为 SCHED_OTHER 的进程，其权值主要取决于两个因素。一个因素是剩下的时间配额，如果用完了则权值为 0。另一个因素是进程的优先级 nice，这是从早期 Unix 沿用下来的负向优先级，其数值表示“谦让”的程度，所以称为“ nice”。其取值范围为 19～-20，以-20 为最高，只有特权用户才能把 nice 值设置成小于 0；而(20 - p->nice)则掉转了它的方向成为 1 至 40。所以，综合的权值在时间配额尚未用完时基本上是二者之和。此外，如果是个内核线程，或者其用户空间与当前进程的相同，因而无需切换用户空间，则会得到一点小“奖励”，将权值额外加 1。

对于实时进程，即调度政策为 SCHED_FIFO 或 SCHED_RR 的进程，则另有一种正向的优先级，

那就是实时优先级 rt_priority，（这里的“rt”表示“real time”），而权值为(1000 + p->rt_priority)。可见，SCHED_FIFO 和 SCHED_RR 两种有时间要求的政策赋予进程很高的权值（相对于 SCHED_OTHER），这种进程的权值至少是 1000。另一方面，rt_priority 的值对于实时进程之间的权值比较也起着重要的作用，其数值也是在系统调用 sched_setscheduler()中与调度政策一起设置的。从这里还可以看出：对于这两种调度实时政策，一个进程已经运行了多久，也就是时间配额 p->counter 的当前值，对权值的计算不起作用。不过，前面已经看到，对于调度政策为 SCHED_RR 的进程，当 p->counter 达到 0 时会导致将进程移到队列的尾部。实时进程的 nice 数值与其优先级无关，但是对 SCHED_RR 进程的时间配额大小有关。由于实时进程的权值有个很大的基数，当有实时进程就绪时非实时进程是没有机会运行的。

由此可见，在 Linux 内核中对权值的计算是很简单的。事实上，在早期的 Unix 系统中实现了一套相当复杂的算法（那时还没有实时进程），后来在实践中觉得那套算法太复杂了，就不断加以简化，在调度的效率、调度的公正性以及其它指标之间反复权衡、折衷，发展成了现在这个样子。另一方面，对实时进程的调度也是 POSIX 标准的要求。不过，goodness()这个函数并不代表 Linux 的调度算法的全部，而要与前面讲到的对 SCHED_RR 的特殊处理、对意欲继续运行的当前进程的特殊处理、以及下面要讲到的 recalculate 结合起来分析。限于篇幅，本书将专注于代码本身的逻辑及过程，而不对调度算法进行定量的分析。

叫到 schedule()。当代码中的 while 循环结束时，变量 c 中的值有几种可能。一种可能是一个大于 0 的正数，此时 next 指向挑选出来的进程。另一种可能是 c 的值为 0，发生于就绪队列中所有进程的权值都是 0 的时候。由于除 init 进程和调用了（系统调用）sched_yield()的进程以外，每个进程的权值最低为 0，所以只要队列中有其它就绪进程存在就不可能为负数。这里要指出，队列中所有其它进程的权值都已降到 0，说明这些进程的调度政策都是 SCHED_OTHER，因为若有政策为 SCHED_FIFO 或 SCHED_RR 的进程存在，则其权值至少也有 1000。

继续往下看（kernel/sched.c）：

==================== kernel/sched.c 578 580 ====================

[schedule()]

```
    /* Do we need to re-calculate counters? */
    if (!c)
        goto recalculate;
```

如果当前已经选择的进程（权值最高的进程）的权值为 0，那就要重新计算各个进程的时间配额。如上所述，这说明系统中当前没有就绪的实时进程。而且，这种情况已经持续了一段时间，因为否则 SCHED_OTHER 进程的权值便没有机会“消耗”到 0。

==================== kernel/sched.c 658 669 ====================

[schedule()]

```
recalculate:
    {
        struct task_struct *p;
        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + NICE_TO_TICKS(p->nice);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            read_unlock(&tasklist_lock);
            spin_lock_irq(&runqueue_lock);
        }
        goto repeat_schedule;

```

宏定义 for_each_task()，读者已经在以前看到过了。这里所作的计算是将每个进程当前的时间配额 p->counter 除以 2，再在上面加上由该进程的 nice 值换算过来的 tick 数量。宏操作 NICE_TO_TICKS 的定义也在前面看到过了（显然，nice 值对于非实时进程既表示优先级也决定着时间配额）。可见，所作的计算是很简单的。这里要注意，for_each_task()是对所有进程的循环，而并不是仅对就绪进程队列的循环。对于不在就绪进程队列中的非实时进程，这里得到了提升其时间配额、从而提升其综合权值的机会。不过，对综合权值的这种提升是很有限的，每次重新计算都将原有的时间配额减半，再与 NICE_TO_TICKS(p->nice)相加，这样就决定了重新计算以后的综合权值永远也不可能达到 NICE_TO_TICKS(p->nice)的两倍。因此，即使经过很长时间的“韬光养晦”，也不能达到可与实时进程竞争的地步（综合权值至少是 1000），所以只是对非实时进程之间的竞争有意义。至于实时进程，时间配额的增加并不会提升其综合权值，而且对于 SCHED_FIFO 进程则连时间配额也是没有意义的。计算完以后，程序转回标号 repeat_schedule 处重新挑选。这样，当再次完成对就绪进程队列的扫描时，变量 c 的值应该不为 0 了，此时 next 指向挑选出来的进程。

进程挑好之后，接下来要做的就是切换的事情了（kernel/sched.c）：

==================== kernel/sched.c 581 596 ====================

[schedule()]

```
        /*
         * from this point on nothing can prevent us from
         * switching to the next task, save this fact in
         * sched_data.
         */
        sched_data->curr = next;
#ifdef CONFIG_SMP
        next->has_cpu = 1;
        next->processor = this_cpu;
#endif
        spin_unlock_irq(&runqueue_lock);

        if (prev == next)
            goto same_process;

#ifdef CONFIG_SMP
```

==================== kernel/sched.c 612 657 ====================

```
#endif /* CONFIG_SMP */

        kstat.context_swtch++;
        /*
         * there are 3 processes which are affected by a context switch:
         *
         * prev == .... ==> (last => next)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
619          *
620          * It's the 'much more previous' 'prev' that is on next's stack,
621          * but prev is set to (the just run) 'last' process by switch_to().
622          * This might sound slightly confusing but makes tons of sense.
623          */
624         prepare_to_switch();
625         {
626                 struct mm_struct *mm = next->mm;
627                 struct mm_struct *oldmm = prev->active_mm;
628                 if (!mm) {
629                         if (next->active_mm) BUG();
630                         next->active_mm = oldmm;
631                         atomic_inc(&oldmm->mm_count);
632                         enter_lazy_tlb(oldmm, next, this_cpu);
633                 } else {
634                         if (next->active_mm != mm) BUG();
635                         switch_mm(oldmm, mm, next, this_cpu);
636                 }
637 
638                 if (!prev->mm) {
639                         prev->active_mm = NULL;
640                         mmdrop(oldmm);
641                 }
642         }
643 
644         /*
645          * This just switches the register state and the
646          * stack.
647          */
648         switch_to(prev, next, prev);
649         __schedule_tail(prev);
650 
651 same_process:
652         reacquire_kernel_lock(current);
653         if (current->need_resched)
654                 goto need_resched_back;
655 
656         return;
657 
```

这里我们跳过对 SMP 结构的条件编译部分。首先，如果挑选出来的进程 next 就是当前进程 prev，就不用切换，直接转到标号 same_process 处就返回了。这里的 reacquire_kernel_lock()对于 i386 单 CPU 结构而言为空语句。前面已经把当前进程的 need_resched 清 0，如果现在又成了非 0 则一定是发生了中断并且情况有了变化，所以转回 tq_scheduler_back 处再调度一次。否则，如果挑选出来的进程 next 与当前进程 prev 不同，那就要切换了。对于 i386 单 CPU 结构而言，prepare_to_switch()也是空语句，而 649 行的__schedule_tail()则只是将当前进程 prev 的 task_struct 结构中 policy 字段里的 SCHED_YIELD 标志位清成 0。所以实际上只剩下了两件事，其一是对用户虚存空间的处理，其二就是进程的切换

PDF-XCHANGE Click to buy NOW! www.docu-track.com

switch_to()。

先来看对用户空间的处理，这里之所以要新开一个 scope 是因为要在堆栈中补充分配两个变量 mm 和 oldmm，前者指向“新进程”next 的 mm_struct 结构，后者则为“老进程”prev 的 active_mm。首先，如果新进程是个不具备用户空间的内核线程，那么其 active_mm 指针也必须是 0，否则就一定是出了问题。但是，内核的设计和实现实际上不允许一个进程（哪怕是内核线程）没有 active_mm，因为指向页面映射目录的指针就在这个数据结构中。所以，如果新进程没有自己的 mm_struct（因此是内核线程），就要在进入运行时向被切换出去的进程借用一个 mm_struct 结构（见 628 行和 630 行）。可是借来的 mm_struct 结构能用吗？能。因为既然没有用户空间，则所需的只是系统空间的映射，而所有进程的系统空间映射都是相同的。那么，借用的 mm_struct 结构什么时候归还呢？到下一次调度其它进程运行时，也就是说当这个内核线程被切换出去时归还，这就是 638 行至 641 行所做的事情。这里的 mmdrop()只是将通过共享借用的 mm_struct 结构中的共享计数减 1，而不是真的将此结构释放，因为这个计数在减 1 以后不可能达到 0。如果新进程 next 有自己的 mm_struct 结构（因此是个进程），那么 next->actieve_mm 必须与 next->mm 相同，否则就有问题了。由于新进程有自己的用户空间，所以就要通过 switch_mm()进行用户空间的切换。这是个 inline 函数，其代码在 include/asm-i386/mmu-context.h 中：

==================== kernel/sched.c 612 657 ====================

[schedule()>switch_mm()]

```
28   static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk, unsigned cpu)
29   {
30        if (prev != next) {
31             /* stop flush ipis for the previous mm */
32             clear_bit(cpu, &prev->cpu_vm_mask);
33             /*
34              * Re-load LDT if necessary
35              */
36             if (prev->context.segments != next->context.segments)
37                  load_LDT(next);
38   #ifdef CONFIG_SMP
39             cpu_tlbstate[cpu].state = TLBSTATE_OK;
40             cpu_tlbstate[cpu].active_mm = next;
41   #endif
42             set_bit(cpu, &next->cpu_vm_mask);
43             /* Re-load page tables */
44             asm volatile("movl %0,%%cr3": :"r" (__pa(next->pgd)));
45        }
46   #ifdef CONFIG_SMP
```

==================== include/asm-i386/mmu_context.h 58 59 ====================

```
58   #endif
59   }
```

对于单 CPU 结构而言，这里关键的语句只有一行，那就是 44 行中的汇编语句，它将新进程页面目录的起始物理地址装入到控制寄存器 CR3 中。我们在第 2 章讲过，CR3 总是指向当前进程的页面目录。至于 LDT 则仅在 VM86 模式中才使用，所以不在我们关心之列。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

读者也许会问：进程本身尚未切换，而存储管理机制中的页面目录指针 CR3 却已经切换了，这样不会造成问题吗？不会的，因为这个时候 CPU 在系统空间运行，而所有进程的页面目录中与系统空间相对应的目录项都指向相同的页面表，所以，不管换上哪一个进程的页面目录都一样，受影响的只是用户空间，系统空间的映射则永远不变。

现在，到了最后要切换进程的关头了。所谓进程的切换主要是堆栈的切换，这是由宏操作 switch_to() 完成的，定义于 include/asm-i386/system.h 中：

==================== include/asm-i386/system.h 15 33 ====================

[schedule()>switch_to()]

```
15   #define switch_to(prev,next,last) do {                              \
16        asm volatile("pushl %%esi\n\t"                                \
17                 "pushl %%edi\n\t"                                   \
18                 "pushl %%ebp\n\t"                                   \
19                 "movl %%esp,%0\n\t"/* save ESP */                   \
20                 "movl %3,%%esp\n\t"/* restore ESP */  \
21                 "movl $1f,%1\n\t"       /* save EIP */              \
22                 "pushl %4\n\t"      /* restore EIP */  \
23                 "jmp __switch_to\n"                \
24                 "1:\t"                             \
25                 "popl %%ebp\n\t"                                    \
26                 "popl %%edi\n\t"                                    \
27                 "popl %%esi\n\t"                                    \
28                 :"=m" (prev->thread.esp),"=m" (prev->thread.eip),   \
29                  "=b" (last)                       \
30                 :"m" (next->thread.esp),"m" (next->thread.eip),     \
31                  "a" (prev), "d" (next),                     \
32                  "b" (prev));                      \
33   } while (0)
```

经历过前面几章中的汇编程序，读者现在对嵌入 C 程序中的汇编语句应该不陌生了。这里的输出部有三个参数，表示这段程序执行以后有三项数据会有改变。其中%0 和%1 都在内存中，分别为 prev->thread.esp 和 prev->thread.eip，而%2 则与寄存器 EBX 结合，对应于参数中的 last。而输入部则有 5 个参数。其中%3 和%4 在内存中，分别为 next->thread.esp 和 next->thread.eip，%5、%6 和%7 分别与寄存器 EAX、EDX 以及 EBX 结合，分别对应于 prev、next 和 prev。

这一段程序虽然只有寥寥数行，却很有奥妙。先来看开头的三条 push 指令和结尾处的三条 pop 指令。看起来好像是很一般，其实却暗藏玄机。且看第 19 行和 20 行。第 19 行将当前的 ESP，也就是当前进程 prev 的系统空间堆栈指针存入 prev->thread.esp，第 20 行又将新受到调度要进入运行的进程 next 的系统空间堆栈指针 next->thread.esp 置入 ESP。这样一来，CPU 在第 20 行与 21 行这两条指令之间就已经切换了堆栈。假定我们有 A、B 两个进程，在本次切换中 prev 指向 A，而 next 指向 B。也就是说，在本次切换中 A 为要“调离”的进程，而 B 为要“切入”的进程。那么，在这里的第 16 至 20 行是在使用 A 的堆栈，而从第 21 行开始就是在用 B 的堆栈了。换言之，从第 21 行开始，“当前进程”，已经是 B 而不是 A 了。我们以前讲过，在内核代码中当需要访问当前进程的 task_struct 结构时使用的指针 current 实际上是宏定义，它根据当前的堆栈指针 ESP 计算出所需的地址。如果第 21 行处引用 current

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的话，那就已经指向 B 的 task_struct 结构了。从这个意义上说，进程的切换在第 20 行的指令执行完就已经完成了。但是，构成一个进程的另一个要素是程序的执行，这方面的切换显然尚未完成。那么，为什么在第 16 至 18 行 push 进 A 的堆栈，而在第 25 行至 27 行却从 B 的堆栈 POP 回来呢？这就是奥妙所在了。其实，第 25 行至 27 行是在恢复新切入的进程在上一次被调离时 push 进堆栈的内容。

那么，程序执行的切换，具体又是怎样实现的呢？让我们来看第 21 行至 24 行。第 21 行将标号“ 1 ”所在的地址，实际上就是第 25 行的 pop 指令所在的地址保存在 prev->thread.eip 中，作为进程 A 下一次被调度运行而切入时的“返回”地址。然后，又将 next->thread.eip 压入堆栈。所以，这里的 next->thread.eip 正是进程 B 上一次被调离时在第 21 行中保存的。它也指向这里的标号“ 1”，即 25 行的 pop 指令。接着，在 23 行通过 jmp 指令，而不是 call 指令，转入了一个函数__switch_to()。且不说在__switch_to()中干了些什么，当 CPU 执行到那里的 ret 指令时，由于是通过 jmp 指令转过去的，最后进入堆栈的 next->thread.eip 就变成了返回地址，而这就是标号“ 1”所在的地址，也就是 25 行的 pop 指令所在的地址。由于每个进程在被调离时都要执行这里的第 21 行，这就决定了每个进程在受到调度恢复运行时都是从这里的第 25 行开始。但是有一个例外，那就是新创建的进程。新创建的进程并没有在“上一次调离时”执行过这里的第 16 至 21 行，所以一来要将其 task_struct 结构中的 thread.eip 事先设置好，二来所设置的“返回地址”也未必是这里的标号“ 1”所在，这取决于其系统空间堆栈的设置。事实上，读者在 fork()一节中已经看到，这个地址在 copy_thread()中（见 arch/i386/kernel/process.c）设置为 ret_from_fork，其代码在 entry.S 中：

==================== arch/i386/kernel/entry.S 179 186 ====================

```
ENTRY(ret_from_fork)
        pushl %ebx
        call SYMBOL_NAME(schedule_tail)
        addl $4, %esp
        GET_CURRENT(%ebx)
        testb $0x02,tsk_ptrace(%ebx)     # PT_TRACESYS
        jne tracesys_exit
        jmp  ret_from_sys_call
```

也就是说，对于新创建的进程，在调用 schedule_tail()以后就直接转到 ret_from_sys_call，“返回 ”到用户空间去了。将前面情景中子进程被创建以后第一次切入时的系统空间堆栈和父进程创建了子进程以后被调度从系统调用 fork()返回而切入时的（系统空间）堆栈作一比较，就可以看得更清楚了，图 4.5 是一幅示意图。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

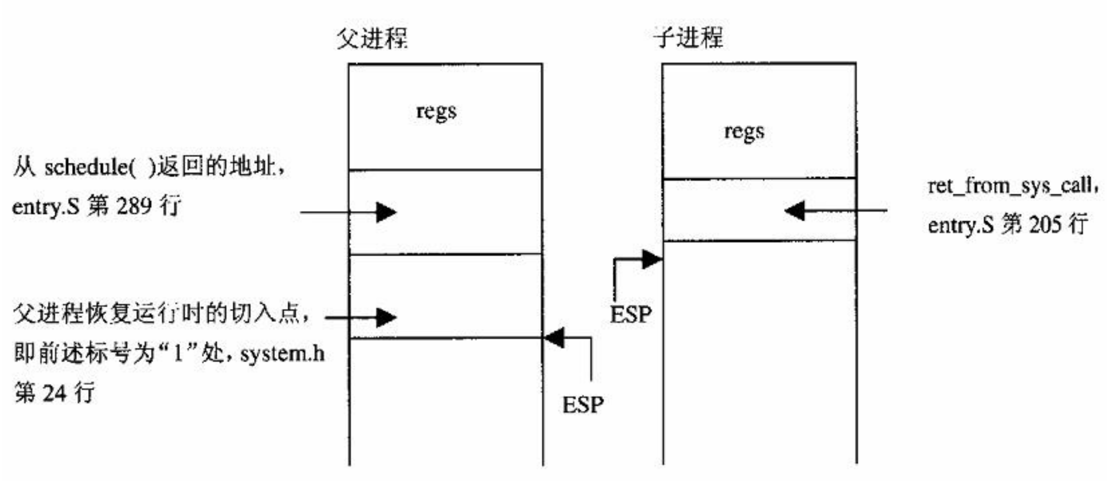


图 4.5　系统调用返回时父子进程系统堆栈对照图

在堆栈空间的顶部，或者说堆栈的“底部”，是父进程因 fork()系统调用而进入系统空间时保存的返回“现场”，包括 CPU 在穿越陷阱门时自动保存在系统空间堆栈中的内容以及通过 entry.S 中的 SAVE_ALL 保存的寄存器内容，合在一起形成一个数据结构 regs。这一部分被原封不动地复制到了子进程的堆栈中，但其中用来返回函数值的 EAX 被设成 0，而指向用户空间堆栈的指针 ESP 也作了相应的修改（见 copy_thread()）。

父进程在 fork()子进程以后，并不立即主动调用 schedule()，而只是将其 task_struct 结构中的 need_resched 标志设成了 1，然后就从 do_fork()和 sys_fork()中返回。经过 entry.S 中的 ret_from_sys_call 到达 ret_with_reschedule 时，如果其 task_struct 结构中的 need_resched 为 0，那就直接返回了，这时其堆栈指针已经指向了 regs，所以 RESTORE_ALL 就使进程回到用户空间（参看第 3 章）。可是，现在 need_resched 已经是 1，就要调用 schedule()进行调度，所以其堆栈指针又回过头来向下伸展。如果调度的结果是继续运行，那就马上会从 schedule()返回，就像什么事也没发生过一样。而如果调度了另一个进程运行，那么其系统空间堆栈就变成了图 4.5 中的样子。处于堆栈“顶部”的是进程在下一次被调度运行时的切入点，那就是在前面 switch_to()的代码中 21 行设置的。注意，switch_to()是一个宏操作而并不是一个函数，所以堆栈中并没有从 switch_to()返回的地址。将来，当父进程被调度恢复运行时，在 switch_to()的 20 行恢复了其堆栈指针，然后在__switch_to()中执行 ret 指令时就“返回”到了 25 行，所以其堆栈中的这一项也可以看成是“从__switch_to()返回的地址”。父进程最后返回到了 entry.S 中的 289 行，紧接着就会跳转到 ret_from_sys_call。相比之下，子进程的这个“返回地址”被设置成 ret_from_sys_call，所以在__switch_to()一执行 ret 指令就直接回到了那里，抄了一小段近路。

最后，在__switch_t()中到底干了些什么呢？看 arch/i386/kernel/process.c 中的相关代码：

==================== arch/i386/kernel/process.c 604 688 ====================

[schedule()>switch_to()>__switch_to()]

```
/*
 *   switch_to(x,yn) should switch tasks from x to y.
 *
 * We fsave/fwait so that an exception goes off at the right time
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
  * (as a call from the fsave or fwait in effect) rather than to
  * the wrong process. Lazy FP saving no longer makes any sense
  * with modern CPU's, and this simplifies a lot of things (SMP
  * and UP become the same).
  *
  * NOTE! We used to use the x86 hardware context switching. The
  * reason for not using it any more becomes apparent when you
  * try to recover gracefully from saved state that is no longer
  * valid (stale segment register values in particular). With the
  * hardware task-switch, there is no way to fix up bad state in
  * a reasonable manner.
  *
  * The fact that Intel documents the hardware task-switching to
  * be slow is a fairly red herring - this code is not noticeably
  * faster. However, there _is_ some room for improvement here,
  * so the performance issues may eventually be a valid point.
  * More important, however, is the fact that this allows us much
  * more flexibility.
  */
void __switch_to(struct task_struct *prev_p, struct task_struct *next_p)
{
     struct thread_struct *prev = &prev_p->thread,
                    *next = &next_p->thread;
     struct tss_struct *tss = init_tss + smp_processor_id();

     unlazy_fpu(prev_p);

     /*
      * Reload esp0, LDT and the page table pointer:
      */
     tss->esp0 = next->esp0;

     /*
      * Save away %fs and %gs. No need to save %es and %ds, as
      * those are always kernel segments while inside the kernel.
      */
     asm volatile("movl %%fs,%0":"=m" (*(int *)&prev->fs));
     asm volatile("movl %%gs,%0":"=m" (*(int *)&prev->gs));

     /*
      * Restore %fs and %gs.
      */
     loadsegment(fs, next->fs);
     loadsegment(gs, next->gs);

     /*
      * Now maybe reload the debug registers
      */
     if (next->debugreg[7]){
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
657             loaddebug(next, 0);
658             loaddebug(next, 1);
659             loaddebug(next, 2);
660             loaddebug(next, 3);
661             /* no 4 and 5 */
662             loaddebug(next, 6);
663             loaddebug(next, 7);
664         }
665 
666         if (prev->ioperm || next->ioperm) {
667             if (next->ioperm) {
668                 /*
669                  * 4 cachelines copy ... not good, but not that
670                  * bad either. Anyone got something better?
671                  * This only affects processes which use ioperm().
672                  * [Putting the TSSs into 4k-tlb mapped regions
673                  * and playing VM tricks to switch the IO bitmap
674                  * is not really acceptable.]
675                  */
676                 memcpy(tss->io_bitmap, next->io_bitmap,
677                      IO_BITMAP_SIZE*sizeof(unsigned long));
678                 tss->bitmap = IO_BITMAP_OFFSET;
679             } else
680                 /*
681                  * a bitmap offset pointing outside of the TSS limit
682                  * causes a nicely controllable SIGSEGV if a process
683                  * tries to use a port IO instruction. The first
684                  * sys_ioperm() call sets up the bitmap properly.
685                  */
686                 tss->bitmap = INVALID_IO_BITMAP_OFFSET;
687         }
688     }
```

这里处理的主要是TSS，其核心就是第638行，将TSS中的内核空间（0级）堆栈指针换成next->esp0。这是因为CPU在穿越中断门或陷阱门时要根据新的运行级别从TSS中取得进程在系统空间的堆栈指针（详见第3章）。其次，段寄存器fs和gs的内容也随后作了相应的切换。至于CPU中为debug而设的一些寄存器，以及说明进程I/O操作权限的位图（见第3章），那就不是我们在这里所关心的了。

我们在第3章中提到过，Intel的原意是让操作系统为每个进程都设置一个TSS，通过切换TSS指针、也就是寄存器TR的内容，由CPU的硬件来实现进程（任务）的切换。表面上看这是很有吸引力的，但是实际上却未必合适。这里，代码的作者加了一段注释，说Linux曾经用过由硬件实现的切换，但后来不用了。注释中说了三个原因，其中第一个原因语焉不详。但是，第二个原因是很有趣的，那就是目前的这种软件实现甚至比硬件实现可以更快。至于第三个原因，即灵活性，那倒是不言而喻的。

总之，除刚创建的新进程外，所有进程在受到调度时的切入点都是在switch_to()（其实是在schedule()中，因为switch_to()是个宏操作）中的标号"1"，一直运行到下一次进入switch_to()以后在__switch_to()中执行ret为止。或者也可以认为，切入点在switch_to()中的25行，一直运行到在下一次进入switch_to()

后的 23 行。总之，这新、旧当前进程的交接点就在 switch_to()这段代码中。

那么，既然都是在同一点上交接，并且从此以后一直到返回用户空间这一段路程又是共同的，不同进程的不同“上下文”又是怎样体现的呢？这不同就在于系统空间堆栈中的内容。不同进程进入系统空间时的运行现场不同，返回地址不同，用户空间堆栈指针不同，一旦回到用户空间就回到了各自的路线上，各奔前程了。

最后，让我们回到在系统调用 exit()中通过 schedule()自愿让出运行的情景（图 4.6）。由于对 schedule()的调用是在 do_exit()中作出的，在交接时这个进程的系统空间堆栈如图 4.6 所示。

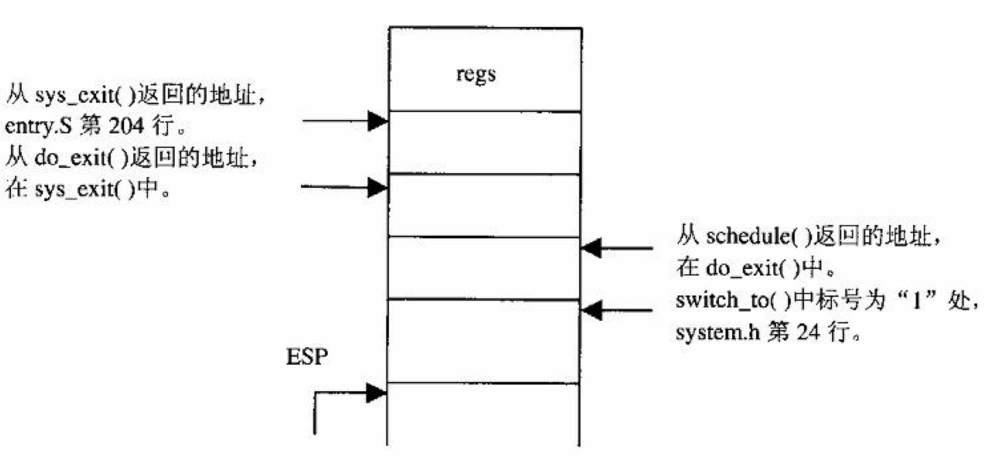


图 4.6　进程切换时系统空间堆栈示意图

从图中可以看出，如果（假定）这个进程像其它进程一样会被调度继续运行的话，它就会循下列的路线返回到用户空间：

(1) 从 switch_to()中的标号“ 1”处恢复运行。由于 switch_to()是宏操作而不是函数，所以这实际上是在 schedule()中。
(2) 从 schedule()返回到 do_exit()中。
(3) 从 do_exit()返回到 sys_exit()中。
(4) 从 sys_exit()返回至到 entry.S 中的 system_call 处，即代码中的 204 行。
(5) 通过宏操作 RESTORE_ALL 回到用户空间。

此处所讲的返回路线与前面讲的系统调用 fork()中的父进程作一比较，可发现，进程主动交出运行时的系统空间堆栈以及返回路线与被动地被剥夺运行时有所不同。前者取决于进程在何处调用 schedule()，而后者则一定是在 entry.S 中的 reschedule 处。

可是，在 exit()这个情景中，由于在调用 schedule()之前已经把进程的状态改成 TASK_ZOMBIE，所以不会再被调度运行。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

4.7 强制性调度

Linux 内核中进程的强制性调度，也就是非自愿的、被动的、剥夺式的调度，主要是由时间引起的。前面讲过，这种调度发生在进程从系统空间返回到用户空间的前夕。当然，并非每次从系统空间返回到用户空间时都会发生此类调度。从第 3 章中以及前节引自 entry.S 的代码片段 ret_with_reschedule 可以看出，此时是否真的调用 schedule()，最终还要取决于当前进程 task_struct 结构中的 need_resched 是否为 1（非 0）。因此，问题就归结为当前进程的 need_resched 是在什么情况下才置成 1 的。在目前版本的内核中，在单 CPU 的条件下，主要有如下几种情况：

- 在时钟中断的服务程序中，发现当前进程（连续）运行的时间过长。
- 当唤醒一个睡眠中的进程时，发现被唤醒的进程比当前进程更有资格运行。
- 一个进程通过系统调用改变调度政策或礼让。这种情况实际上应该被视为主动的、自愿的调度，因此这样的系统调用会引起立即调度。

先看第一种情况。在前一节，读者已看到，调度时要为可执行进程队列（就绪进程队列）中的每个进程都计算出一个当时的权值。对于一般交互式的应用，其数值主要取决于进程剩下的时间配额，即 task_struct 结构中的一个计数器 counter 的当前值。对于有实时要求的进程，也就是调度政策为 SCHED_RR 或 SCHED_FIFO 的进程，则运行资格与此无关，并且都有非常高的权值。当队列中所有的进程均为交互式进程，即调度政策为 SCHED_OTHER 的进程，并且所有这些进程都用完了时间配额时，就要重新计算并设置每个进程的时间配额，其数值主要取决于为各个进程设定的优先级别。在运行中，则每当发生时钟中断时都要递减当前进程的时间配额，使当前进程的运行资格逐渐降低，当计数器的值降至 0 时，就要强制进行一次调度，剥夺当前进程的运行。

在第 3 章的“时钟中断”一节中，读者已经看到，在时钟中断服务程序 do_timer_interrupt()中要调用一个函数 do_timer()，并已浏览过这个函数的代码。在这个函数中，对于单 CPU 结构（在 SMP 结构中各个 CPU 使用本地的定时器，称为 APIC 定时器）要调用另一个函数 update_process_times()来调整当前进程与时间有关的一些运行参数，其代码在 kemel/sched.c 中：

==================== kernel/timer.c 575 597 ====================
[do_timer_interrupt()>do_timer()>update_process_times()]

```
/*
 * Called from the timer interrupt handler to charge one tick to the current
 * process.  user_tick is 1 if the tick is user time, 0 for system.
 */
void update_process_times(int user_tick)
{
	struct task_struct *p = current;
	int cpu = smp_processor_id(), system = user_tick ^ 1;

	update_one_process(p, user_tick, system, cpu);
	if (p->pid) {
		if (--p->counter <= 0) {
			p->counter = 0;
			p->need_resched = 1;
```

```
        }
        if (p->nice > 0)
            kstat.per_cpu_nice[cpu] += user_tick;
        else
            kstat.per_cpu_user[cpu] += user_tick;
        kstat.per_cpu_system[cpu] += system;
    } else if (local_bh_count(cpu) || local_irq_count(cpu) > 1)
        kstat.per_cpu_system[cpu] += system;
}
```

只要不是 0 号进程，就从当前进程的计数器中减 1。当计数器降到 0 时，就将 task_struct 结构中的 need_resched 置成 1。至于函数中其它的操作，包括 update_one_process()，只是与统计信息有关，我们在这里并不关心，读者可以自己阅读。

再看第二种情况。在内核种，当要唤醒一个睡眠中的进程时，可以调用一个函数 wake_up_process()。这个函数的代码也在 kernel/sched.c 中：

==================== kernel/sched.c 321 344 ====================

```
/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
 * progress), and as such you're allowed to do the simpler
 * "current->state = TASK_RUNNING" to mark yourself runnable
 * without the overhead of this.
 */
inline void wake_up_process(struct task_struct * p)
{
    unsigned long flags;

    /*
     * We want the common case fall through straight, thus the goto.
     */
    spin_lock_irqsave(&runqueue_lock, flags);
    p->state = TASK_RUNNING;
    if (task_on_runqueue(p))
        goto out;
    add_to_runqueue(p);
    reschedule_idle(p);
out:
    spin_unlock_irqrestore(&runqueue_lock, flags);
}
```

可见，所谓“唤醒”，就是把进程的状态设置成 TASK_RUNNING，并将进程挂入 runqueue（即可执行进程队列），然后便调用函数 reschedule_idel()。对于单 CPU 结构来说，这个函数很简单。

==================== kernel/sched.c 198 207 ====================

```
/*
 * This is ugly, but reschedule_idle() is very timing-critical.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * We are called with the runqueue spinlock held and we must
 * not claim the tasklist_lock.
 */
static FASTCALL(void reschedule_idle(struct task_struct * p));

static void reschedule_idle(struct task_struct * p)
{
#ifdef CONFIG_SMP
```

===================== kernel/sched.c 286 294 =====================

```
#else /* UP */
	int this_cpu = smp_processor_id();
	struct task_struct *tsk;

	tsk = cpu_curr(this_cpu);
	if (preemption_goodness(tsk, p, this_cpu) > 1)
		tsk->need_resched = 1;
#endif
}
```

其目的是将所唤醒的进程与当前进程之间作一比较。如果所唤醒的进程运行资格更高就将当前进程的 need__rescbed 标志设置成 1。函数 preemption_goodness()计算两个进程综合权值的差，其代码也是在 kernel/sched.c 中定义的：

===================== kernel/sched.c 189 196 =====================

[wake_up_process()>reschedule_idle()>preemption_goodness()]

```
/*
 * the 'goodness value' of replacing a process on a given CPU.
 * positive value means 'replace', zero or negative means 'dont'.
 */
static inline int preemption_goodness(struct task_struct * prev, struct task_struct * p, int cpu)
{
	return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->active_mm);
}
```

读者也许注意到了，在 reschedule_idle()中的当前进程指针并不是通过宏操作 current 而是通过另一个宏操作 cpu_curr 得到的。这两者有什么区别呢？先来看看 cpu_curr 的定义，那也是在 kernel/sched.c 中定义的：

===================== kernel/sched.c 103 103 =====================

```
#define cpu_curr(cpu) aligned_data[(cpu)].schedule_data.curr
```

不知读者是否记得，这是在 schedule()中挑选了要进入运行但尚未切换之前设置的，（见 kernel/sched.c，586 行）。所以，在大部分时间中这是与 current 一致的，但是在完成切换之前的一小段时间里，这个进程并不是真正的当前进程。可是，将刚唤醒的进程与这个进程相比显然更准确，因为当 CPU 要从系统空间返回到用户空间时，这个进程已经“在位”了。

第三种情况，实际上应该被视为自愿的让出。但是，从内核代码的形式上看，也是通过相同的办法，将当前进程的 need_resched 标志置为 1，使得在进程返回用户空间前夕发生调度，所以也放在这一

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

节中。此类系统调用有两个，一个是 sched_setscheduler()，另一个是 sched_yield()。系统调用 sched_setscheduler()的作用是改变进程的调度政策。用户登录到系统后，第一个进程的适用调度政策为 SCHED_OTHER，也就是默认为无实时要求的交互式应用。在通过 fork()创建新进程时则将此进程适用的调度政策遗传给了子进程。但是，用户可以通过系统调用 sched_setscheduler()改变其适用调度政策。内核代码中对此系统调用的实现 sys_sched_setscheduler()在 kernel/sched.c 中：

==================== kernel/sched.c 957 966 ====================

```
asmlinkage long sys_sched_setscheduler(pid_t pid, int policy,
                        struct sched_param *param)
{
    return setscheduler(pid, policy, param);
}

asmlinkage long sys_sched_setparam(pid_t pid, struct sched_param *param)
{
    return setscheduler(pid, -1, param);
}
```

==================== kernel/sched.c 887 955 ====================

```
static int setscheduler(pid_t pid, int policy,
            struct sched_param *param)
{
    struct sched_param lp;
    struct task_struct *p;
    int retval;

    retval = -EINVAL;
    if (!param || pid < 0)
        goto out_nounlock;

    retval = -EFAULT;
    if (copy_from_user(&lp, param, sizeof(struct sched_param)))
        goto out_nounlock;

    /*
     * We play safe to avoid deadlocks.
     */
    read_lock_irq(&tasklist_lock);
    spin_lock(&runqueue_lock);

    p = find_process_by_pid(pid);

    retval = -ESRCH;
    if (!p)
        goto out_unlock;

    if (policy < 0)
        policy = p->policy;
    else {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
917             retval = -EINVAL;
918             if (policy != SCHED_FIFO && policy != SCHED_RR &&
919                         policy != SCHED_OTHER)
920                   goto out_unlock;
921       }
922
923       /*
924        * Valid priorities for SCHED_FIFO and SCHED_RR are 1..99, valid
925        * priority for SCHED_OTHER is 0.
926        */
927       retval = -EINVAL;
928       if (lp.sched_priority < 0 || lp.sched_priority > 99)
929             goto out_unlock;
930       if ((policy == SCHED_OTHER) != (lp.sched_priority == 0))
931             goto out_unlock;
932
933       retval = -EPERM;
934       if ((policy == SCHED_FIFO || policy == SCHED_RR) &&
935           !capable(CAP_SYS_NICE))
936             goto out_unlock;
937       if ((current->euid != p->euid) && (current->euid != p->uid) &&
938           !capable(CAP_SYS_NICE))
939             goto out_unlock;
940
941       retval = 0;
942       p->policy = policy;
943       p->rt_priority = lp.sched_priority;
944       if (task_on_runqueue(p))
945             move_first_runqueue(p);
946
947       current->need_resched = 1;
948
949 out_unlock:
950       spin_unlock(&runqueue_lock);
951       read_unlock_irq(&tasklist_lock);
952
953 out_nounlock:
954       return retval;
955 }
```

从代码中可以看出，Linux 内核有三种不同的调度政策，即 SCHED_FIFO、SCHED_RR 以及 SCHED_OTHER。每个进程都必然采取其中之一（见 918 行）。除调度政策外还有一些参数，一个进程的调度政策与调度参数结合在一起就决定了它受内核调度的种种特性。

这里的 capable()是个 inline 函数，它检查 current->cap_effective，看某个标志位是否为 1，也就是进程是否允许进行某种特定的操作。文件 include/inux/capability.h 中定义了所有的标志位。函数 move_first_runqueue()将进程从可执行进程队列的当前位置移到队列的前部（如果该进程在可执行进程队列中的话），使其在调度时（相对于具有相同运行资格的进程）处于较为有利的地位。最后将当前进

PDF-XCHANGE Click to buy NOW! www.docu-track.com

程的 need_resched 设成 1，强制发生一次调度。

另一个系统调用 sched_yield()使运行中的进程可以为其它进程“让路”，但并不进入睡眠。内核中的实现 sys_sched_yield()也在 kernel/sched.c 中：

==================== kernel/sched.c 1019 1052 ====================

```
asmlinkage long sys_sched_yield(void)
{
    /*
     * Trick. sched_yield() first counts the number of truly
     * 'pending' runnable processes, then returns if it's
     * only the current processes. (This test does not have
     * to be atomic.) In threaded applications this optimization
     * gets triggered quite often.
     */

    int nr_pending = nr_running;

#if CONFIG_SMP
    int i;

    // Substract non-idle processes running on other CPUs.
    for (i = 0; i < smp_num_cpus; i++)
        if (aligned_data[i].schedule_data.curr != idle_task(i))
            nr_pending--;
#else
    // on UP this process is on the runqueue as well
    nr_pending--;
#endif
    if (nr_pending) {
        /*
         * This process can only be rescheduled by us,
         * so this is safe without any locking.
         */
        if (current->policy == SCHED_OTHER)
            current->policy |= SCHED_YIELD;
        current->need_resched = 1;
    }
    return 0;
}
```

与改变调度政策或参数时不同的是，这里并不改变当前进程在可执行进程队列中的位置。不言而喻，“礼让”只有在系统中还有其它就绪进程存在的情况下才有意义，所以这里先要检查 nr_pending，即正在等待运行的进程数量。代码中将 current->policy 中的 SCHED_YIELD 标志置为 1，这个标志位在紧接着的调度中就清成 0。有关的代码在__schedule_tail()中，这是在 schedule()中通过 switch_to()切换进程以后调用的。

与主动调度相比，通过将当前进程的 need_resched 标志置 1 以强制进行的调度有一个重要的不同，那就是从发现有调度的必要到调度真正发生有个延迟，叫做“调度延迟（dispatch latency）”。在前列的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

三种条件中，第三种（改变调度政策或礼让）对时间并不敏感。第一种虽是由时间引起，但实际上也并无实时的要求。而第二种，就是当唤醒一个进程并发现该进程的权值比当前进程更高，也就是更为紧急时，这就有时间要求了。

唤醒一个睡眠中的进程有两种来源。一种是进程间通信，例如一个进程向另一个进程发送了一个信号，读者已经在系统调用 exit()一节中看到过。进程间通信当然不局限于信号发送，其它的例子如以后读者在“进程间通信”中会看到的通过管道、报文队列以及 socket 等手段进行的通信。典型的情景就是在 client/server 方式的应用中。一个进程在睡眠中等待来自其它进程的服务请求，而当其它进程通过某种手段向其发送一个请求时就要将其从睡眠中唤醒。而第二种来源则通常更为紧急，那就是某个事件的发生引起了一次中断，在中断服务程序中或 bh 函数中由于该事件的发生而要将某个进程（或若干个进程）唤醒，使其可以在用户空间对事件作进一步的处理。这种情况往往有更高的时间要求。这里有两个问题，第一是当调度发生时被唤醒的进程是否一定会被挑选上。这一点由于 SCHED_FIFO 和 SCHED_RR 两种调度政策的设立和优先级的使用而有了保证。第二就是到底什么时候（多少微秒之内）会发生调度，这一点在目前的 linux 内核中是没有保证的，而只能从统计的、平均的角度看是否能满足条件。

4.8　系统调用 nanosleep()和 pause()

出于种种原因，运行中的进程常常需要主动进入睡眠状态，并发起一次调度让出 CPU。这一定要通过系统调用，或者在系统调用内部才能做到。注意，前一节中讲到的系统调用 sched_yield()与此有所不同，那只是让内核进行一次调度，而当前进程继续保持可运行状况。而这里所说的是，当前进程进入睡眠，也就是将进程的状态变成 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE，并从可执行队列中脱钩，调度的结果一定是其它进程得以运行。并且，进程一旦进入睡眠状态，就需要经过唤醒才能将状态恢复成 TASK_RUNNING，并回到可执行队列中。

这种主动在一段时间内放弃运行、让出 CPU 的行动可以分成两种。一种是隐含的，不确定的，就是说暂时让出 CPU 的可能性隐含在其它行为之中。此时让出 CPU 本身并不是目的，而只是在真正的目的一时不能达到，必须等待时才出于公德心，把 CPU 暂时让出来。这样的例子有 read()、write()、open()、send()、recvfrom()等等，几乎所有与外设有关的系统调用都有可能在执行的过程中受阻而进入睡眠、让出 CPU。另一种是明确的，目的就在进入睡眠状态。这样的系统调用主要有两个，一个是 nanosleep()，另一个是 pause()。

系统调用 nanosleep()使当前进程进入睡眠状态，但是在指定的时间以后由内核将该进程唤醒，所以常常用来实现周期性的应用。程序员们常常使用的 sleep()是个库函数，实际上是通过系统调用 nanosleep()来实现的。

系统调用 pause()也使当前进程进入睡眠，可是与时间无关，要到接收到一个信号时才被唤醒，所以常常用来协调若干进程的运行。读者在前几节中看到的系统调用 wait4()，类似的还有 wait3()，实际上可以看作是 pause()的一种特例，因为它要在接收到特定的信号 SIGCHLD 并且满足若干特殊条件时才被唤醒。

还有一种特殊情况，当前进程接收到了信号 SIGSTOP，然后在当前进程从系统空间返回到用户空

PDF-XCHANGE Click to buy NOW! www.docu-track.com

间之前（不管是因为系统调用、中断或是异常），就会在 do_signal()中调用 schedule()，进程状态变成 TASK_STOPPED，并从可执行队列中脱钩，一直要到收到一个 SIGCONT 信号时才能恢复到可运行状态。这种情况实际上是强制性的，但由于在形式上当前进程在 do_signal()的过程中“主动”调用 schedule()，所以没有把它放在强制性调度一节中，我们在讲进程间通信时还要讲到这个话题。

这一节中我们集中介绍 nanosleep()和 pause()两个系统调用。

系统调用 nanosleep()在内核中的实现为 sys_nanosleep()，其代码在 kernel/sched.c 中：

==================== kernel/timer.c 797 836 ====================

```
asmlinkage long sys_nanosleep(struct timespec *rqtp, struct timespec *rmtp)
{
	struct timespec t;
	unsigned long expire;

	if(copy_from_user(&t, rqtp, sizeof(struct timespec)))
		return -EFAULT;

	if (t.tv_nsec >= 1000000000L || t.tv_nsec < 0 || t.tv_sec < 0)
		return -EINVAL;


	if (t.tv_sec == 0 && t.tv_nsec <= 2000000L &&
	    current->policy != SCHED_OTHER)
	{
		/*
		 * Short delay requests up to 2 ms will be handled with
		 * high precision by a busy wait for all real-time processes.
		 *
		 * Its important on SMP not to do this holding locks.
		 */
		udelay((t.tv_nsec + 999) / 1000);
		return 0;
	}

	expire = timespec_to_jiffies(&t) + (t.tv_sec || t.tv_nsec);

	current->state = TASK_INTERRUPTIBLE;
	expire = schedule_timeout(expire);

	if (expire) {
		if (rmtp) {
			jiffies_to_timespec(expire, &t);
			if (copy_to_user(rmtp, &t, sizeof(struct timespec)))
				return -EFAULT;
		}
		return -EINTR;
	}
	return 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

库函数 sleep()的参数是以秒为单位的整数，而 nanosleep()的参数则为两个 timespec 结构指针。第一个指针 rqtp，指向给定所需睡眠时间的数据结构；第二个指针 rmtp，则指向返回剩余睡眠时间的数据结构。这是因为睡眠中的进程有可能因接收到信号而提前被唤醒，这时候函数返回-1 并在 rmtp 所指的数据结构中返回剩余的时间（如果 rmtp 不是 NULL），然后进程可以决定是否再次睡眠把时间用光。

数据结构 timespec 的定义在 include/linux/time.h 中：

==================== include/linux/time.h 9 12 ====================

```
struct timespec {
	time_t	tv_sec;		/* seconds */
	long	tv_nsec;	/* nanoseconds */
};
```

这里的 tv_sec，单位为秒，而 tv_nsec 为毫微秒，也就是 10^{-9} 秒。当然，这并不表示睡眠时间的精度可以达到毫微秒的量级。以前讲过，在典型的内核配置中时钟中断的频率 Hz 为 100（见 include/asm-i386/param.h），也就是说时钟中断的周期为 10 毫秒。这意味着，如果进程进入睡眠而循正常途径由时钟中断服务程序来唤醒的话，那就只能达到 10 毫秒的精度。正因为这样，才有 809～821 行的特殊处理，那就是如果要求睡眠的时间小于 2 毫秒，而要求睡眠的进程又是个有实时要求的进程（其调度政策为 SCHED_FIFO 或 SCHED_RR），那就不能真的让这个进程进入睡眠，因为那样有可能要到 10 毫秒以后才能将其唤醒，对于实时应用的进程来说这是不能接受的。所以，在这样的情况下能提供的只是延迟而不是睡眠。这里由一个宏操作 udelay()通过计数来实现延迟，其定义在 include/asm-i386/delay.h 中：

==================== include/asm-i386/delay.h 16 18 ====================

```
#define udelay(n) (__builtin_constant_p(n) ? \
	((n) > 20000 ? __bad_udelay() : __const_udelay((n) * 0x10c6ul)) : \
	__udelay(n))
```

除若干预定的常数以外，都是通过函数__udelay()完成延迟，其代码在 arch/i386/lib/delay.c 中。我们把涉及的各个函数逐层列在下面，供读者阅读：

==================== arch/i386/lib/delay.c 76 79 ====================

[sys_nanosleep()>udelay()>__udelay()]

```
void __udelay(unsigned long usecs)
{
	__const_udelay(usecs * 0x000010c6);  /* 2**32 / 1000000 */
}
```

下 ysnanodeeP()>u 血 lay()>udday()>consLudday()1

==================== arch/i386/lib/delay.c 67 74 ====================

```
inline void __const_udelay(unsigned long xloops)
{
	int d0;
	__asm__("mull %0"
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        :"=d" (xloops), "=&a" (d0)
        :"1" (xloops),"0" (current_cpu_data.loops_per_jiffy));
    __delay(xloops * HZ);
}
```

常量 current_cpu_data.loops_per_sec 的数值取决于具体的 CPU，系统初始化时由内核根据采集的数据确定，并保存在数据结构 current_cpu_data 中：

==================== arch/i386/lib/delay.c 59 65 ====================

[sys_nanosleep()>udelay()>__udelay()>__const_udelay()>__delay()]

```
void __delay(unsigned long loops)
{
    if(x86_udelay_tsc)
        __rdtsc_delay(loops);
    else
        __loop_delay(loops);
}
```

如果 CPU 支持基于硬件的延迟，那么就通过__rdtsc_delay()完成所需的延迟，否则由软件通过计数实现。

==================== arch/i386/lib/delay.c 42 57 ====================

[sys_nanosleep()>udelay()>__udelay()>__const_udelay()>__delay()>__loop_delay()]

```
/*
 *  Non TSC based delay loop for 386, 486, MediaGX
 */

static void __loop_delay(unsigned long loops)
{
    int d0;
    __asm__ __volatile__(
        "\tjmp 1f\n"
        ".align 16\n"
        "1:\tjmp 2f\n"
        ".align 16\n"
        "2:\tdecl %0\n\tjns 2b"
        :"=&a" (d0)
        :"0" (loops));
}
```

读者对于嵌入 C 代码的汇编语句已经不陌生了，所以这里不再解释。从这段代码中可以看出，udelay()是通过计数循环来达到延迟的。也就是说，这种情况下当前进程并不真的进入睡眠，并不让出 CPU，而只是通过循环来消磨掉一些时间。这当然不是个好办法，但对于有实时要求的进程也只好不得已而为之。再说，即使对于有实时要求的进程，只要延迟的时间超过 2 毫秒，也不用通过这种办法来实现。可是，为什么会有这么短的延迟要求呢？这一般是与外设操作相联系的，有些外设要求连续两次操作之间的时间间隔不得小于某个特定值，所以就有了这么短的延迟要求。

回到 sys_nanosleep()的代码中。对于正常的睡眠要求，先调用 timespec_to_jiffies()，将数据结构 t

PDF-XCHANGE Click to buy NOW! www.docu-track.com

中的数值换算成时钟中断的次数，换算的方法在 time.h 中，我们把它留给读者自己阅读（time.h）：

==================== include/linux/time.h 17 42 ====================

[sys_nanosleep()>timespec_to_jiffies()]

```
/*
 * Change timeval to jiffies, trying to avoid the
 * most obvious overflows..
 *
 * And some not so obvious.
 *
 * Note that we don't want to return MAX_LONG, because
 * for various timeout reasons we often end up having
 * to wait "jiffies+1" in order to guarantee that we wait
 * at _least_ "jiffies" - so "jiffies+1" had better still
 * be positive.
 */
#define MAX_JIFFY_OFFSET ((~0UL >> 1)-1)

static __inline__ unsigned long
timespec_to_jiffies(struct timespec *value)
{
     unsigned long sec = value->tv_sec;
     long nsec = value->tv_nsec;

     if (sec >= (MAX_JIFFY_OFFSET / HZ))
          return MAX_JIFFY_OFFSET;
     nsec += 1000000000L / HZ - 1;
     nsec /= 1000000000L / HZ;
     return HZ * sec + nsec;
}
```

注意，前面 sys_nanosleep()中的 822 行的(t.tv_sec || t.tv_nsec)是关系表达式，其值为 1 或者 0。

然后，将当前进程的状态改为 TASK_INTERRUPT 并调用 schedule_timeout()进入睡眠。以前讲过，睡眠状态 TASK_INTERRUPT 与 TASK_UNINTERRUPT 的区别在于后者在进程接收到信号时不会被唤醒。函数 schedule_timeout()的代码也在 sched.c 中：

==================== kernel/sched.c 369 419 ====================

[sys_nanosleep()>schedule_timeout()]

```
signed long schedule_timeout(signed long timeout)
{
     struct timer_list timer;
     unsigned long expire;

     switch (timeout)
     {
     case MAX_SCHEDULE_TIMEOUT:
          /*
           * These two special cases are useful to be comfortable
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
             * in the caller. Nothing more. We could take
             * MAX_SCHEDULE_TIMEOUT from one of the negative value
             * but I' d like to return a valid offset (>=0) to allow
             * the caller to do everything it want with the retval.
             */
            schedule();
            goto out;
        default:
            /*
             * Another bit of PARANOID. Note that the retval will be
             * 0 since no piece of kernel is supposed to do a check
             * for a negative retval of schedule_timeout() (since it
             * should never happens anyway). You just have the printk()
             * that will tell you if something is gone wrong and where.
             */
            if (timeout < 0)
            {
                printk(KERN_ERR "schedule_timeout: wrong timeout "
                      "value %lx from %p\n", timeout,
                      __builtin_return_address(0));
                current->state = TASK_RUNNING;
                goto out;
            }
        }

        expire = timeout + jiffies;

        init_timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);

        timeout = expire - jiffies;

    out:
        return timeout < 0 ? 0 : timeout;
}
```

在内核中把时钟中断的次数作为计时的统一尺度，并给时钟中断之间的间隔起了个名字叫做“ jiffy ”（“瞬间”的意思）。与此相应，内核中设置了一个全局的计数器 jiffies，用来对系统自初始化以来时钟中断的次数计数。所以，在调用 schedule_timeout()之前把需要睡眠的时间先换算成时钟中断的数量，把这个数量与当前的 jiffies 相加就得到了“到点”的时间。但是，当所要求的时间太长，长到不能用带符号整数表达时（其实是最大的正数减 1，见前面 sys_nanosleep()函数代码中对 timespec_to_jiffies()的注解以及代码中的第 822 行），就返回一个常数 MAX_JIFFY_OFFSET。这个常数在 schedule_timeout()

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

中被视作“无限期”，所以在 384 行中调用 schedule()就完事了。既然是无限期睡眠，内核就不承担按时将其唤醒的责任，这个进程要一直睡眠到有另一个进程向其发送一个信号时才会被唤醒。

函数 schedule_timeout()的返回值是进程被唤醒时剩下的还未睡完的时间。我们来看看当调用参数为 MAX_JIFFY_OFFSET 时的返回值。在这种情况下，当进程被唤醒而从 schedule()返回时就通过 goto 语句转到标号 out 处，而变量 timeout 的数值在这整个过程中并未改变，仍旧是 MAX_JIFFY_OFFSET，这体现了从无限减去有限后结果还是无限的原理。

当要求的睡眠时间在规定的范制以内时，内核就要承担起按时将此进程唤醒的责任了。为此目的，内核要设置好一个“定时器”，也就是这里的数据结构 timer，并将其挂入一个定时器队列，而每次时钟中断时都要检查这些定时器是否到点。数据结构 timer 的类型为 timer_list，是在 include/linux/time.h 中定义的，详见第 3 章中的“时钟中断”一节。我们在那里提到了这个数据结构及其作用，但没有深入加以讨论，这是因为那时我们还没有讲过进程调度及有关的机制，很难真正讲清楚。而现在，结合 schedule_timeout()的代码，就可以把整个过程和机制讲清楚了。这里，在 init_timer()以后，将定时器的到点时间设置成计算得到的 expire。到点时要执行的函数则为 process_timeout()，等一下我们就会看到它到底干些什么了。准备传给 process_timeout()的参数为 current，读者应该还记得，这实际上是一个得到当前进程 task_struct 指针的宏操作。读者也许会问，为什么不干脆把数据结构中的变量 data 改成 task_struct 指针？这是因为这样更为灵活、通用，再说到点时要调用的函数也并不总是与某个进程直接有关的。

函数 add_timer()将 timer 挂入定时器队列，其代码在 timer.c 中：

==================== kernel/timer.c 176 190 ====================

[sys_nanosleep()>schedule_timeout()>add_timer()]

```
void add_timer(struct timer_list *timer)
{
	unsigned long flags;

	spin_lock_irqsave(&timerlist_lock, flags);
	if (timer_pending(timer))
		goto bug;
	internal_add_timer(timer);
	spin_unlock_irqrestore(&timerlist_lock, flags);
	return;
bug:
	spin_unlock_irqrestore(&timerlist_lock, flags);
	printk("bug: kernel timer added twice at %p.\n",
			__builtin_return_address(0));
}
```

核心的操作是在 internal_add_timer()完成的，这里多了一层“包装”，目的是将核心的队列操作保护起来。由 spin_lock_irqsave()先将中断关闭，而 spin_unlock_irqsave()则在操作以后再恢复原状。函数 internal_add_timer()的代码还是在同一文件中（timer.c）：

==================== kernel/timer.c 122 160 ====================

[sys_nanosleep()>schedule_timeout()>add_timer()>internal_add_timer()]

```
static inline void internal_add_timer(struct timer_list *timer)
```

```
123 {
124 	/*
125 	 * must be cli-ed when calling this
126 	 */
127 	unsigned long expires = timer->expires;
128 	unsigned long idx = expires - timer_jiffies;
129 	struct list_head * vec;
130 
131 	if (idx < TVR_SIZE) {
132 		int i = expires & TVR_MASK;
133 		vec = tv1.vec + i;
134 	} else if (idx < 1 << (TVR_BITS + TVN_BITS)) {
135 		int i = (expires >> TVR_BITS) & TVN_MASK;
136 		vec = tv2.vec + i;
137 	} else if (idx < 1 << (TVR_BITS + 2 * TVN_BITS)) {
138 		int i = (expires >> (TVR_BITS + TVN_BITS)) & TVN_MASK;
139 		vec =  tv3.vec + i;
140 	} else if (idx < 1 << (TVR_BITS + 3 * TVN_BITS)) {
141 		int i = (expires >> (TVR_BITS + 2 * TVN_BITS)) & TVN_MASK;
142 		vec = tv4.vec + i;
143 	} else if ((signed long) idx < 0) {
144 		/* can happen if you add a timer with expires == jiffies,
145 		 * or you set a timer to go off in the past
146 		 */
147 		vec = tv1.vec + tv1.index;
148 	} else if (idx <= 0xffffffffUL) {
149 		int i = (expires >> (TVR_BITS + 3 * TVN_BITS)) & TVN_MASK;
150 		vec = tv5.vec + i;
151 	} else {
152 		/* Can only get here on architectures with 64-bit jiffies */
153 		INIT_LIST_HEAD(&timer->list);
154 		return;
155 	}
156 	/*
157 	 * Timers are FIFO!
158 	 */
159 	list_add(&timer->list, vec->prev);
160 }
```

在 128 行中引用的 timer_jiffies 也是个全局量，表示当前对定时器队列的处理在时间上已经推进到了哪一点，同时也是设置定时器的基准点，其数值有可能会不同于 jiffies，等一会儿我们就会看到它的作用。

在进一步深入到 internal_add_timer()的代码中去之前，有必要先大致介绍一下定时器队列的组织。本来，最简单的办法是将所有的 timer_list 结构，即定时器，按“到点”的先后链接在一起成为一个队列，然后每当 jiffies 改变时就从该队列的头部开始逐个检查并处理这些数据结构，直到发现第一个尚未到点的定时器时就可以结束了。可是这样有个缺点，就是每当要将一个新的定时器加入到这个队列中去时，要在队列中进行线性搜索，寻找适当的链入位置，在最坏的情况下要扫描过队列中所有的数

PDF-XCHANGE Click to buy NOW! www.docu-track.com

据结构。当队列中的成员数量有可能很大时，这种方案的效率就不能令人满意了。学过数据结构与算法的读者可能马上会想到可以通过“杂凑”（hash）来改善效率。也就是说，将这些定时器数据结构组织成一个队列数组，或者说队列的阵列，而不是一个单一的队列，然后根据每个定时器到点的时间经过杂凑计算决定应该将其链入到哪一个队列中。这样，通过将定时器分散链入到不同的队列中，就可以减小各个队列的平均长度，从而提高效率。最简单的杂凑计算莫过于从数值中抽取最低的若干位，也就是通过“与”运算将数值中的高位屏蔽掉，这实际上相当于将数值除以一个 2 的整数次幂以后取其余数。但是，在这种简单的杂凑表组织里每个队列中还会有很多分属于不同到点时间的定时器，这是因为只要杂凑计算后的结果相同就会被链入到同一个队列中。例如，jiffies 是个 32 位无符号整数，假如我们取最低的 10 位作为杂凑计算的结果，也就是说在数组中有 2^{10} 个队列，那么从理论上说在最坏的情况下在一个队列中可以有分属于 2^{22} 种不同到点时间的定时器。当然，在实际运行中是不会这么糟糕的，但是总叫人觉得不尽如人意。理想的解决方案是每个队列中只有属于同一到点时间的定时器。可是总不可能设置 2^{32} 个定时器队列吧？所以，既要顾及在时钟中断发生时检查并处理这些定时器的效率，又要顾及在将定时器插入到这些队列中去时的效率，对此机制的设计和实现是一种挑战。Linux 内核比较好地解决了这个问题，设计并实现了一种相当巧妙的方案。

在 Linux 内核中设置了五个而不是一个这样的杂凑表，即定时器队列数组。详见下列代码（timer.c）：

==================== kernel/timer.c 74 102 ====================

```
/*
 * Event timer code
 */
#define TVN_BITS 6
#define TVR_BITS 8
#define TVN_SIZE (1 << TVN_BITS)
#define TVR_SIZE (1 << TVR_BITS)
#define TVN_MASK (TVN_SIZE - 1)
#define TVR_MASK (TVR_SIZE - 1)

struct timer_vec {
     int index;
     struct list_head vec[TVN_SIZE];
};

struct timer_vec_root {
     int index;
     struct list_head vec[TVR_SIZE];
};

static struct timer_vec tv5;
static struct timer_vec tv4;
static struct timer_vec tv3;
static struct timer_vec tv2;
static struct timer_vec_root tv1;

static struct timer_vec * const tvecs[] = {
     (struct timer_vec *)&tv1, &tv2, &tv3, &tv4, &tv5
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
102  };
```

数据结构 tv1、tv2、…、tv5 每个都包含了一个 timer_list 指针数组，这就是所谓杂凑表（bucket），表中的每个指针都指向一个定时器队列。其中 tv1 与其它几个数据结构的不同仅在于数组的大小，tv1 中的数组大小为 2^8，而其它几个的大小都是 2^6。这样，队列的数量总共是 $2^8+4\times2^6=512$，还是可以接受的。每个数组都与一个变量 index 相联系，用来指示当下一个时钟中断发生时要处理的队列。与此同时，将 32 位的到点时间也划分成五段，其中最低的一段为 8 位，与 tv1 相对应，其它四段则都是 6 位。要将一个定时器挂入队列中去时，先根据到点时间和当前时间计算出这个定时器应该在多少次时钟中断以后到点，如果这个差值小于 256 的话就取到点时间的最低 8 位作为其杂凑值，然后用这个杂凑值作为下标在 tv1 的数组中找到相应的队列，并将此定时器链入到这个队列中。由于 tv1 的数组中有 256 个队列，所以每个队列中的定时器都具有相同的到点时间。可是，当差值大于等于 256 时怎么办呢？这时候就看差值是否小于 2^{14}，如果是，就取到点时间的数值中的第二段（6 位，从第 8 位至第 14 位）为杂凑值，或下标，并将定时器插入到 tv2 的某个队列中去。示意图如图 4.7。

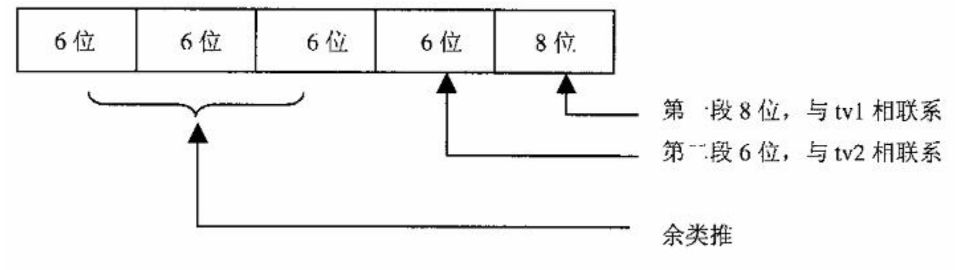


图 4.7　定时器队列数组下标确定规则示意图

显然，tv2 中的队列与 tv1 中的不同，因为 tv1 中每个队列里的定时器都属于同一个到点时间，而 tv2 中的队列则不然。理论上 tv2 中的每个队列都可能含有分属 256 个不同到点时间的定时器。也就是说，tv2 的“尺度”与 tv1 不同。当差值大于 2^{14} 时，那就要进一步看差值是否大于 2^{20} 了，余类推。

现在可以回到 internal_add_timer()的代码中了。读者应该可以自己读懂这段代码，其中具体将定时器链入到队列中的操作由 list_add()完成。

也就是说，每次都是插入到队列的尾部。对于 tv1 中的队列来说，由于每个队列中所有的定时器都是在同一时间到点，所以插入的位置根本没有关系；而对于其它的队列来说，下面就会看到其实也没有关系。这样，将一个定时器链入到队列中的操作变得很简单，根本就不需要在队列中寻找合适的插入位置了，从而其代价成了一个常数，而与队列长度无关了。同时，当时钟中断发生，从而将 jiffies 向前推进一步时，只要在 tv1 中根据 index 的指示将一个队列中所有的定时器都处理一遍（执行定时器所指定的函数）并将这些定时器释放，然后将 index 也向前推进一步就行了。当 tv1.index 达到 256 时就又将其设成 0，回到数组的开头，开始另外一轮的 256 次时钟中断。此时，由于一个 tv1 周期已经完成，就从 tv2 中根据 tv2.index 的指引将 tv2 中的一个队列搬运到 tv1 中。在搬运的过程中，对队列中的每个定时器都再调用一次 internal_add_timer()。此时该队列中所有定时器的到点时间与当前时间的差都已小于 256（由于当前时间的推进），所以都会被分散到 tv1 中的各个队列中去，而与各个定时器在队列中的位置无关。由此可见，链入 tv2 各个队列里的定时器是分两步到位进入 tv1 中的队列（第一步进入 tv2，第 2 步进入 tv1）。依次类推，当到点时间与当前时间的差大于 2^{26} 时要先进入 tv5，分五步才能

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进入 tv1。虽然有些定时器要分好几步才能到达 tv1 中，其代价仍然与队列长度无关，并且有个上限，就是最多五步。所以，这个办法要比线性搜索好得多。

将定时器链入到某个队列中以后，schedule_timeout()就调用 schedule()，使当前进程真正地进入睡眠，等待唤醒。

那么，时钟中断怎样唤醒这个进程呢？

在第 3 章中的“时钟中断”一节中，我们看到在从时钟中断返回之前要执行与时钟有关的 bh 函数 timer_bh()，而 timer_bh()要调用一个函数 run_timer_list()（kernel/timer.c）：

==================== kernel/timer.c 668 672 ====================

```
void timer_bh(void)
{
	update_times();
	run_timer_list();
}
```

函数 run_timer_list()的代码在 kernel/sched.c 中：

==================== kernel/timer.c 288 324 ====================

[timetbh()＞runnmerlist()]

```
	struct task_struct *tsk;

	tsk = cpu_curr(this_cpu);
	if (preemption_goodness(tsk, p, this_cpu) > 1)
		tsk->need_resched = 1;
#endif
}

/*
 * Careful!
 *
 * This has to add the process to the _beginning_ of the
 * run-queue, not the end. See the comment about "This is
 * subtle" in the scheduler proper..
 */
static inline void add_to_runqueue(struct task_struct * p)
{
	list_add(&p->run_list, &runqueue_head);
	nr_running++;
}

static inline void move_last_runqueue(struct task_struct * p)
{
	list_del(&p->run_list);
	list_add_tail(&p->run_list, &runqueue_head);
}

static inline void move_first_runqueue(struct task_struct * p)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
      list_del(&p->run_list);
      list_add(&p->run_list, &runqueue_head);
}

/*
 * Wake up a process. Put it on the run-queue if it's not
 * already there.  The "current" process is always on the
 * run-queue (except when the actual re-schedule is in
```

在“时钟中断”一节中，我们还讲过，在特殊的情况下 jiffies 向前推进的步长有可能大 1。正因为这样，这里通过一个循环来处理 jiffies 的每个单步。在每个单步中，先看 tv1.index 是否为 0，若为 0 就要从 tv2 中搬运一个队列到 tv1 中。我们也把这种情况暂时搁一下，先来看不为 0 时的情况。

代码中由 goto 实现的循环就是处理在这一步中到点的队列。处理本身是很简单的，顺着队列挨个把定时器通过 detach_timer()从队列中摘除出来，然后就执行该定时器所指定的函数。执行完这整个队列时，就将 times_jiffies 和 tv1.index 也往前推进一步。但是，tv1.index 的值是以 256 为模的（TVR_MASK），所以其数值在 255 以后就回到了 0，下一个循环中或者下一次执行这个函数时就要通过 cascade_timers()从 tv2 中搬运一个队列到 tv1 中来。tv2 中也有一个 index，也要向前推进。每当 jiffies 向前推进了 256 步，也就是每当发生了 256 次时钟中断时，tv2.index 就要往前推进一步。与 tv1.index 不同，tv2.index 是以 64 为模的，所以在达到 63 以后就要回到 0。当 tv2.index 为 1 时就要从 tv3 中搬运一个队列到 tv2 中和 tv1 中，余类推。

为什么是在 tv2.index 为 1 时，而不是为 0 时，才从 tv3 中搬运呢？回头去看一下 internal 的代码就清楚了。当到点时间与当前时间的差 idx 为 TVR_SIZE 即 256 时，经过第 136 行的处理以后结果为 1 而不是 0。实际上，tv2 中下标为 0 的那个队列一定是空的。同时，为了便于实现，代码中将 tv1、tv2 等五个数据结构也放在一个数组中，这就是 tvecs[]。这里将下标设成从 1 开始，就是表示从 tv2 开始搬运，而第 298 行则表示如果 tv2.index 推进以后变成了 1 就要进一步从 tv3 搬运，余类推。

这里的 NOOF_TVECS 为一常数，实际上就是 5（timer.c）：

==================== kernel/timer.c 104 104 ====================

```
#define NOOF_TVECS (sizeof(tvecs) / sizeof(tvecs[0]))
```

函数 cascade_timers()的代码也在同一文件中。这是一段简单的代码，我们就不加解释了。

==================== kernel/timer.c 264 286 ====================

[timer_bh()>run_timer_list()>cascade_timers()]

```
static inline void cascade_timers(struct timer_vec *tv)
{
      /* cascade all the timers from tv up one level */
      struct list_head *head, *curr, *next;

      head = tv->vec + tv->index;
      curr = head->next;
      /*
       * We are removing _all_ timers from the list, so we don't  have to
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
          * detach them individually, just clear the list afterwards.
          */
         while (curr != head) {
              struct timer_list *tmp;

              tmp = list_entry(curr, struct timer_list, list);
              next = curr->next;
              list_del(curr); // not needed
              internal_add_timer(tmp);
              curr = next;
         }
         INIT_LIST_HEAD(head);
         tv->index = (tv->index + 1) & TVN_MASK;
    }
```

在我们这个情景中，定时器中的函数指针为 process_timeout，参数为睡眠中进程的 task_struct 指针，所以到点时就会调用 process_timeout()（sched.c）：

==================== kernel/sched.c 362 367 ====================
[timer_bh()>run_timer_list()>process_timeout()]

```
static void process_timeout(unsigned long __data)
{
     struct task_struct * p = (struct task_struct *) __data;

     wake_up_process(p);
}
```

函数通过 wake_up_process 将睡眠中的进程唤醒。它的代码读者已经在前一节“强制性调度”中看到过了。进程被唤醒并且再次被调度运行时，就回到了前面的 schedule_timeout()中。换句话说，是该进程从前面 schedule_timeout()中的 schedule()返回了。

回过去继续看 schedute_timeout()的代码，从 schedule()返回以后紧接着就调用了 del_timer_sync()，读者也许会感到奇怪，刚才在 run_timer_list()中不是已经通过 detach_timer()把定时器从队列中删除了吗？怎么这里又要 del_timer_sync()呢？对于单处理器的系统，del_timer_sync()定义为 del_timer()，我们来看看 detach_timet()和 del_timer()的代码（sched.c）：

==================== kernel/timer.c 192 198 ====================
[timer_bh()>run_timer_list()>detach-tiner()]

```
static inline int detach_timer (struct timer_list *timer)
{
     if (!timer_pending(timer))
          return 0;
     list_del(&timer->list);
     return 1;
}
```

==================== include/linux/timer.h 54 57 ====================
［tinsewbhh()＞runtimerlist()＞detacLtlner()＞timeL 卫 ending()］

```
long time_esterror = NTP_PHASE_LIMIT;/* estimated error (us)      */
```

```
long time_phase;            /* phase offset (scaled us) */
long time_freq = ((1000000 + HZ/2) % HZ - HZ/2) << SHIFT_USEC;
                            /* frequency offset (scaled ppm)*/
```

所以 detach_tiner()仅在所处理的 timer_list 数据结构在队列中时才把它从队列中删除。函数 del_timer()实际上调用 detach_timer():

==================== kernel/timer.c 213 223 ====================

[sys_nanosleep()>schedute_timeout()>del_timer()]

```
int del_timer(struct timer_list * timer)
{
	int ret;
	unsigned long flags;

	spin_lock_irqsave(&timerlist_lock, flags);
	ret = detach_timer(timer);
	timer->list.next = timer->list.prev = NULL;
	spin_unlock_irqrestore(&timerlist_lock, flags);
	return ret;
}
```

可见，对一个已经从队列中脱链的定时器再调用一次 del_timer()并没有害处。可是，即使没有害处，也没有理由做无用功啊。是的，但是要想到，run_timer_list()并不是惟一可以将这个进程唤醒的函数。当另一个进程向睡眠中的进程发送一个信号时，同样可以将其唤醒。所以，在 schedule_timcout()中再调用一次 del_timer()就可以确保安全了。这里要指出，这里的 timer 是个局部量，其空间在堆栈中，一旦从 schedule_timeout()返回，这个数据就消失了。这里可以省去动态分配和释放缓冲器的麻烦，也可以提高效率。可是将这样一个数据结构留在队列中是很危险的，一定要保证在这个数据结构还有效时将其从队列中去除。

最后，期望中的到点时间 expire 与当前时间 jiffies 之差为剩下的尚未睡够的时间。这剩下的尚未睡够的时间是以时钟中断的次数为尺度的，所以在 sys_nanosheep()中又将其换算回 timespec 数据结构中的秒和毫微秒，然后返回给用户空间。当然，只有在进程因信号而被唤醒时才有可能还未睡够。否则，睡过了头的可能倒是有的。这一方面是因为在特殊的情况下也许会把好几次时钟中断合并在一起进行对 jiffies 的处理，所以一次就向前推进好几步。另一方面即使按时将进程唤醒也不能保证该进程马上就会被调度运行。

系统调用 sys_nanoshecp()并非 schedule_timeout()的惟一“用户”。内核中还提供了一个函数 interruptible_sleep_on_timeout()，供各种设备驱动程序在内核中使用，将来在设备驱动一章中读者会看到它的使用。此外，住内核中也可以直接调用 schedule_timeout()。

与 sys_nanosheep()相比，同样也是系统调用的 sys_pause()的代码就很简单了，其代码在 arch/i386/kernel/sys_i386.c 中：

==================== arch/i386/kernel/sys_i386.c 250 255 ====================

```
asmlinkage int sys_pause(void)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
	current->state = TASK_INTERRUPTIBLE;
	schedule();
	return -ERESTARTNOHAND;
}
```

显然，当前进程通过 sys_pause()入睡以后，只有在接收到信号时才会被唤醒。

4.9　内核中的互斥操作

内核中的很多操作在进行的过程中都不容许受到打扰，最典型的例子就是队列操作。如果两个进程都要将一个数据结构链入到同一个队列的尾部，要是在第一个进程完成了一半的时候发生了调度，让第二个进程插了进来，结果很可能就乱了。类似的干扰也有可能来自某个中断服务程序或 bh 函数。在多处理器 SMP 结构的系统中，这种干扰还有可能来自另一个处理器。

不过，除了一个进程主动调用 schedule()让出 CPU 的情况（显然不会发生在不容许受到打扰的过程中途）之外，只有在从系统空间返回到用户空间的前夕才有可能发生调度。这样的安排使得上述两个进程间的干扰实际上不会发生在内核中。这一点在“进程的调度与切换”一节中已经讨论过了。所以，上述两个进程在内核中互相干扰的情况实际上只会发生在多处理器的系统中，在单处理器的系统中是不会发生的。但是，在另一种情况下，则仍有可能发生进程间的干扰。系统中有些资源是共享的，但是在具体使用期间却需要独占，而且对这些资源的访问可能会受阻而需要睡眠等待。进程在访问此类资源的时候就可能受到其它进程的干扰。至于来自中断服务程序（包括 bh 函数）的干扰，则总是有可能的。并且，在多处理器系统中，不但要防止来自同一处理器上的中断服务程序的干扰，还要防止来自其它处理器的中断服务程序的干扰。

在“多处理器 SMP 系统结构”一章中，我们将讨论有关多处理器结构的种种问题。但是，如上所述，如果不加防止，单处理器系统中在一定条件下也会发生进程间的互相干扰。另一方面，这种措施也被借用在系统调用 vfork()中，用作父进程与子进程之间对于共享虚存空间的互斥保护手段。

进程间对共享资源的互斥访问，或者说对进程间干扰的防范，是通过“信号量”（semaphore）这种机制来实现的。内核中为此提供了 down()和 up()两个函数，分别对应于操作系统理论中的 P 和 V 两种操作。至于信号量，则是一种数据结构类型 semaphore。

先看数据结构，struct semaphore 是在 include/asm/i386/semaphore.h 中定义的：

==================== include/asm-i386/semaphore.h 44 51 ====================

```
struct semaphore {
	atomic_t count;
	int sleepers;
	wait_queue_head_t wait;
#if WAITQUEUE_DEBUG
	long __magic;
#endif
};
```

计数器 count 所计的就是“信号量”中的那个“量”，它代表着可使用资源的数量。没有学习过操

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

作系统理论的读者不妨把这个数据结构想像成一个院子的大门，而 count 表示一共有几张门票。当一个进程想要进入这个院子的围墙里面干些什么时，先要在大门口领取门票。所以 count 的数值即表示还有几个进程可以进门。在典型的情况下，一共就只有一张票，所以只有一个进程可以进去。

当一个进程来到门口要领票，却发现门票已经发完的时候，就只好到大门旁边的“休息室”去睡眠、等候。这个休息室就是这里的队列 wait，而计数器 sleepers 则表示有几个进程正在等候。进入了院子的进程，完成了它要做的事情以后，还是从同一个大门出来，并将门票交还。如果在交还门票之前，门票的数量已经是 0，那就可能有进程正在休息室中等候，所以还要向这些正在等候的进程打个招呼，说“现在有门票了”，或者说，将这些进程唤醒，让它们去竞争那张门票。可见，原理其实很简单。下面通过一段实例，看看这段过程具体是怎么实现的。这段实例取自系统调用 umount()，我们在这里并不关心怎样拆卸（umount）一个已经安装的文件系统，而是关心怎样把一部分关键性的操作保护起来。

下面的代码取自文件 fs/super.c：

==================== fs/super.c 44 50 ====================

```
44   /*
45    * We use a semaphore to synchronize all mount/umount
46    * activity - imagine the mess if we have a race between
47    * unmounting a filesystem and re-mounting it (or something
48    * else).
49    */
50   static DECLARE_MUTEX(mount_sem);
```

==================== fs/super.c 1117 1118 ====================

```
1117 asmlinkage long sys_umount(char * name, int flags)
1118 {
```

==================== fs/super.c 1144 1146 ====================

```
1144      down(&mount_sem);
1145      retval = do_umount(nd.mnt, 0, flags);
1146      up(&mount_sem);
```

==================== fs/super.c 1152 1153 ====================

```
1152      return retval;
1153 }
```

这里的目的是要把 do_umount()保护起来，因为在同一时间里整个系统中只允许有一个进程在安装或拆卸文件系统，而安装或拆卸文件系统的过程又是可能（实际上是必定）受阻，因而中途会发生调度的。为达到这个目的，首先在第 50 行建立起一个独门的“院子”，或者说“信号量 ” mount_sem，并且把要加以保护的操作放在进门（down）和出门（up）两个操作之间。要进入这个“院子”时必须要先执行 down()以得到一张“门票”，而当完成了操作从里面出来时则要执行 up()以归还门票并唤醒可能正在等待的其它进程。操作系统理论里把这段需要独家关起门来干的操作称为“临界区”（critical section）。顺便说一下，把 critical section 翻译成“临界区”似乎有点学究气，critical 其实就是“非常重要，搞不好的话后果可能很严重”的意思。

有关 DECLAR_MUTEX()的定义在 include/asm-i386/semaphore.h 中：

==================== include/asm-i386/semaphore.h 53 71 ====================

```
53   #if WAITQUEUE_DEBUG
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
54  # define __SEM_DEBUG_INIT(name) \
55           , (int)&(name).__magic
56  #else
57  # define __SEM_DEBUG_INIT(name)
58  #endif
59
60  #define __SEMAPHORE_INITIALIZER(name,count) \
61  { ATOMIC_INIT(count), 0, __WAIT_QUEUE_HEAD_INITIALIZER((name).wait) \
62       __SEM_DEBUG_INIT(name) }
63
64  #define __MUTEX_INITIALIZER(name) \
65       __SEMAPHORE_INITIALIZER(name,1)
66
67  #define __DECLARE_SEMAPHORE_GENERIC(name,count) \
68       struct semaphore name = __SEMAPHORE_INITIALIZER(name,count)
69
70  #define DECLARE_MUTEX(name) __DECLARE_SEMAPHORE_GENERIC(name,1)
71  #define DECLARE_MUTEX_LOCKED(name) __DECLARE_SEMAPHORE_GENERIC
```

宏定义 ATOMIC_INIT()和__WAIT_QUEUE_HEAD_INITIALIZER()分别在 include/asm-i386/atomic.h 和 include/linux/wait.h 中，读者可以自行参阅。总之，经过 gcc 的预处理以后，前面的第 50 行就变成类似于这样的语句：

```
static struct semaphore  mount_sem={[(1)], 0, ...}
```

也就是说，通过 DECLARE_MUTEX()建立的信号量只有 1 张“门票”，所以只有一个进程可以进入临界区。另一种通过 DECLARE_MUTEX_LOCKED()建立的信号量则一张门票也没有，一定要等到某个进程通过 up()操作送来一张才能把它发给一个进程而允许其进入大门。读者已经在系统调用 fork()一节中看到过此种信号量的运用。两种信号量各有各的用处，而 MUTEX 和 MUTEX_LOCKED 正反映了它们各自的用途。此外，信号量既可以作为全局量存在，也可以作为某个函数的局部量存在。

对于信号量的操作只有 down()和 up()两种，这是两个 inline 函数，都是在 include/asm-i386/semaphore.h 中定义的。先看 down()：

===================== include/asm-i386/semaphore.h 109 132 =====================

```
109  /*
110   * This is ugly, but we want the default case to fall through.
111   * "__down_failed" is a special asm handler that calls the C
112   * routine that actually waits. See arch/i386/kernel/semaphore.c
113   */
114  static inline void down(struct semaphore * sem)
115  {
116  #if WAITQUEUE_DEBUG
117       CHECK_MAGIC(sem->__magic);
118  #endif
119
120       __asm__ __volatile__(
121            "# atomic down operation\n\t"
```

```
122             LOCK "decl %0\n\t"     /* --sem->count */
123             "js 2f\n"
124             "1:\n"
125             ".section .text.lock,\"ax\"\n"
126             "2:\tcall __down_failed\n\t"
127             "jmp 1b\n"
128             ".previous"
129             :"=m" (sem->count)
130             :"c" (sem)
131             :"memory");
132  }
```

这段嵌入汇编代码的输出部为空，说明执行后并不改变寄存器的内容；而输入部则使指针 sem 与寄存器 ECX 结合。由于 count 是 semaphore 数据结构中的第一个成分，所以指向该数据结构的指针 sem 即为指向 sem->count 的指针，从而第 122 行的 decl 指令所递减的实际上是 sem->count。减了以后的结果若为 0 或大于 0，或者说如果成功地拿到了一张门票，那么就在标号“1”处结束了。注意这里在指令 decl 前面有个前缀 LOCK，表示在执行这条指令时要把总线锁住，以防可能来自同一系统中其它 CPU 的干扰。

如果减了以后的结果为负数，那就表示拿不到门票，就转到标号“2”处调用__down_failed()。实际上，进程在__down_failed()中会进入睡眠，一直要到被唤醒并成功地拿到门票才会从那里返回，然后转到标号“1”而结束 down()操作，即进入了临界区。

函数__down_failed()以及有关的代码都在 arch/i386/kernel/semaphore.c 中：

==================== arch/i386/kernel/semaphore.c 171 193 ====================

[down()>__down_failed()]

```
/*
 * The semaphore operations have a special calling sequence that
 * allow us to do a simpler in-line version of them. These routines
 * need to convert that sequence back into the C sequence when
 * there is contention on the semaphore.
 *
 * %ecx contains the semaphore pointer on entry. Save the C-clobbered
 * registers (%eax, %edx and %ecx) except %eax when used as a return
 * value..
 */
asm(
".align 4\n"
".globl __down_failed\n"
"__down_failed:\n\t"
     "pushl %eax\n\t"
     "pushl %edx\n\t"
     "pushl %ecx\n\t"
     "call __down\n\t"
     "popl %ecx\n\t"
     "popl %edx\n\t"
     "popl %eax\n\t"
     "ret"
```

```
);
```

显然，这里的目的只在于调用__down()。代码的作者在这个文件（semaphore.c）的开头处加了一段注释，或可帮助读者更好地理解：

==================== arch/i386/kernel/semaphore.c 20 39 ====================

```
/*
 * Semaphores are implemented using a two-way counter:
 * The "count" variable is decremented for each process
 * that tries to acquire the semaphore, while the "sleeping"
 * variable is a count of such acquires.
 *
 * Notably, the inline "up()" and "down()" functions can
 * efficiently test if they need to do any extra work (up
 * needs to do something only if count was negative before
 * the increment operation.
 *
 * "sleeping" and the contention routine ordering is
 * protected by the semaphore spinlock.
 *
 * Note that these functions are only called when there is
 * contention on the lock, and as such all this is the
 * "non-critical" part of the whole semaphore business. The
 * critical part is the inline stuff in <asm/semaphore.h>
 * where we want to avoid any extra jumps and calls.
 */
```

再看__down()的代码：

==================== arch/i386/kernel/semaphore.c 58 89 ====================

[down()>__down_failed()__down()]

```
void __down(struct semaphore * sem)
{
     struct task_struct *tsk = current;
     DECLARE_WAITQUEUE(wait, tsk);
     tsk->state = TASK_UNINTERRUPTIBLE;
     add_wait_queue_exclusive(&sem->wait, &wait);

     spin_lock_irq(&semaphore_lock);
     sem->sleepers++;
     for (;;) {
          int sleepers = sem->sleepers;

          /*
           * Add "everybody else" into it. They aren't
           * playing, because we own the spinlock.
           */
          if (!atomic_add_negative(sleepers - 1, &sem->count)) {
               sem->sleepers = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
76                  break;
77              }
78              sem->sleepers = 1; /* us - see -1 above */
79              spin_unlock_irq(&semaphore_lock);
80
81              schedule();
82              tsk->state = TASK_UNINTERRUPTIBLE;
83              spin_lock_irq(&semaphore_lock);
84          }
85          spin_unlock_irq(&semaphore_lock);
86          remove_wait_queue(&sem->wait, &wait);
87          tsk->state = TASK_RUNNING;
88          wake_up(&sem->wait);
89      }
```

有关等待队列中各元素的数据结构 wait_queue_t 以及宏定义 DECLARE_WAITQUEUE()，读者已在前一节中看到过，此处不再赘述。而 add_wait_queue_exclusive()则把代表当前进程的等待队列元素 wait 链入到由队列头 sem->wait 代表的等待队列的尾部。当 CPU 执行到达 for(;;)循环时，sem->sleepers 表明（连当前进程在内）一共有几个进程正在等待着要进入临界区。另一方面，虽然当前进程是因为拿不到“门票”，进不了临界区才到了__down()中，但是由于在这里的 spin_lock_irq()之前并没有加锁，说不定已经有某个进程（当然是在另一个处理器上）在此期间已经执行了一次 up()操作，因而这个时候实际上已经有“门票”了。如果不再作一次检查，那就会无谓地进入睡眠而等待已经存在的“门票”。更糟的是，可能再也没有进程会来唤醒它了。所以，在 for()循环中通过 atomic_ddd_negative()所作的检查是很关键的。而且，它所作的还不仅仅是检查，它将(sleepers-1)加到 sem->count 上去，使得它的值不会小于-1。举例来说，如果在当前进程执行 down()之前 sem->count 为 0，并且从那时候以来并无进程在此信号量上执行 up()，那么(sleepers-1)为 0，而 sem->count 为-1，相加的结果仍是-1，此时 atomic_add_negative()返回非零，表示当前进程仍需等待。而若在 65 行之前已经有个进程在此信号量上执行了 up()操作，那么 sleepers-1 仍为 0，但 sem->count 变成了 0，相加的结果为 0 而不是负数，此时 atomic_add_negative()返回零，表示当前进程不需要等待了，可以进入临界区了，就好像本次操作在 down()里面将 sem->count 从 1 变成了 0 一样。当 sem->count 的值为正数或 0 时表示还有多少资源，或者可以理解为还剩下几张“门票”; 而 sem->count 为负的时候则表明已经没有资源并且有进程正在等待，却并不需要表明到底有几个进程正在等待。这样，在 up()操作中可以用一条指令将 sem->count 加 1，然后根据结果是否为 0 判定是否有进程需要唤醒。

如果当前进程发现不再需要等待了，它就通过这里的 break 语句跳出 for 循环，并在返回之前唤醒等待队列中的其它进程。不过，如果有其它进程正在等待的话，被唤醒之后多半通不过第 74 行的测试，这是因为那时候 sem->sleepers 已经设成了 0（见第 75 行），所以(sleepers-1)为-1，已经是个负数；除非那时 sem->count 已经变成 1，否则 atomic_add_negative()必然会返回非 0。这一点等一下我们还要讨论。

当 atomic_add_negative()返回非 0 时，当前进程就真的要进入睡眠状态等待了，所以在第 81 行调用 schedule()。由于入睡的状态为 TASK_UNINTERRUPTIBLE，所以不会因接收到信号而被唤醒。同时，由于标志位 TASK_EXCLUSIVE 为 1，所以只有排在队列中的第一个进程才会被唤醒。还要指出的是，当睡眠中的进程被唤醒而从 schedule()中返回，并回到循环体的前部时，由于 sem->sleepers 在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

78 行被设成 1，所以此时(sleepers-1)必然为 0，所以能否进入临界区的条件取决于当时的 sem->count 是否为负数(-1)。在典型的情况下，被 up()操作所唤醒的进程会碰上 sem->count 为 0，从而能跳出 for()循环从__down()返回而进入临界区。在从__down()返回之前还要再从队列中唤醒一个进程，而那个进程就往往要继续等待了。

从代码中可以看出，当有多个进程在等待进入一个临界区时，当前进程略有些优势，然后就是“先来先进”，而进程的优先级别并没有起作用。在有实时要求的系统中这未必不是一个缺陷，将来的版本中也许会考虑这个问题。

还有个问题也与优先级和临界区相联系，称为“优先级倒转”。试想这么一种情景：一个优先级很高的进程在某一临界区门外等待，而正在临界区里面的进程偏偏优先级很低，而且一旦因操作受阻进入睡眠，然后被唤醒时便一时得不到机会运行，于是便“急惊风遇上了慢郎中”。在这样的情况下，优先级高的进程因临界区内的进程优先级太低而受了连累。解决的方法是，当有优先级高的进程在临界区外等待的时候，就暂时把它的高优先级“借”给临界区内的进程，提高其竞争力。目前在 Linux 内核中尚未实现此种机制，这也是一个可以改进的地方。不过，问题也并不像想像中那么严重，因为内核中需要在临界区内进行的操作一般都是很短促的，不至于受阻；反之，必须在临界区内进行、而又有可能中途受阻而需要睡眠的操作，则一般不宜由优先级很高的进程来进行。

再来看 up()就比较简单了，这也是在 semaphore.h 中：

==================== include/asm-i386/semaphore.h 182 205 ====================

```
/*
 * Note! This is subtle. We jump to wake people up only if
 * the semaphore was negative (== somebody was waiting on it).
 * The default case (no contention) will result in NO
 * jumps for both down() and up().
 */
static inline void up(struct semaphore * sem)
{
#if WAITQUEUE_DEBUG
     CHECK_MAGIC(sem->__magic);
#endif
     __asm__ __volatile__(
          "# atomic up operation\n\t"
          LOCK "incl %0\n\t"     /* ++sem->count */
          "jle 2f\n"
          "1:\n"
          ".section .text.lock,\"ax\"\n"
          "2:\tcall __up_wakeup\n\t"
          "jmp 1b\n"
          ".previous"
          :"=m" (sem->count)
          :"c" (sem)
          :"memory");
}
```

显然，与 down()的代码是相似的，不同之处仅在于这是递增，而不是递减 sem->count，并且在递

PDF-XCHANGE Click to buy NOW! www.docu-track.com

增以后结果为 0 或负数时就调用__up_wakeup()，那也是在 semaphore.c 中：

==================== arch/i386/kernel/semaphore.c 219 231 ====================
[up()>__up_wakeup()]

```
asm(
".align 4\n"
".globl __up_wakeup\n"
"__up_wakeup:\n\t"
     "pushl %eax\n\t"
     "pushl %edx\n\t"
     "pushl %ecx\n\t"
     "call __up\n\t"
     "popl %ecx\n\t"
     "popl %edx\n\t"
     "popl %eax\n\t"
     "ret"
);
```

同样，__up()的代码也在 semaphore.c 中：

==================== arch/i386/kernel/semaphore.c 41 54 ====================
[up()>__up_wakeup()>__up()]

```
/*
 * Logic:
 *  - only on a boundary condition do we need to care. When we go
 *    from a negative count to a non-negative, we wake people up.
 *  - when we go from a non-negative count to a negative do we
 *    (a) synchronize with the "sleeper" count and (b) make sure
 *    that we're on the wakeup list before we synchronize so that
 *    we cannot lose wakeup events.
 */

void __up(struct semaphore *sem)
{
     wake_up(&sem->wait);
}
```

这里的 wake_up()和一些有关的宏定义都是在 sched.h 中定义的：

==================== include/linux/sched.h 555 560 ====================

```
#define wake_up(x)          __wake_up((x),TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
#define wake_up_all(x)           __wake_up((x),TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE,0)
#define wake_up_sync(x)          __wake_up_sync((x),TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
#define wake_up_interruptible(x) __wake_up((x),TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
#define wake_up_interruptible_all(x) __wake_up((x),TASK_INTERRUPTIBLE,0)
#define wake_up_interruptible_sync(x)__wake_up_sync((x),TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
```

而__wake_up()则在 sched.c 中。读者可以看到这个函数依次唤醒一个队列中的所有符合条件的进

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

程。但是，如果一个被唤醒进程的 TASK_EXCLUSIVE 标志为 1 就不再继续唤醒队列中其余的进程了（kernel/sched.c）。

==================== kernel/sched.c 766 769 ====================

[up()>__up_wakeup()>__up()>wake_up()>__wake_up()]

```
void __wake_up(wait_queue_head_t *q, unsigned int mode, unsigned int wq_mode)
{
	__wake_up_common(q, mode, wq_mode, 0);
}
```

==================== kernel/sched.c 692 764 ====================

[up()>__up_wakeup()>__up()>wake_up()>__wake_up()>__wake_up_common()]

```
static inline void __wake_up_common (wait_queue_head_t *q, unsigned int mode,
				     unsigned int wq_mode, const int sync)
{
	struct list_head *tmp, *head;
	struct task_struct *p, *best_exclusive;
	unsigned long flags;
	int best_cpu, irq;

	if (!q)
		goto out;

	best_cpu = smp_processor_id();
	irq = in_interrupt();
	best_exclusive = NULL;
	wq_write_lock_irqsave(&q->lock, flags);

#if WAITQUEUE_DEBUG
	CHECK_MAGIC_WQHEAD(q);
#endif

	head = &q->task_list;
#if WAITQUEUE_DEBUG
	  if (!head->next || !head->prev)
		WQ_BUG();
#endif
	tmp = head->next;
	while (tmp != head) {
		unsigned int state;
		    wait_queue_t *curr = list_entry(tmp, wait_queue_t, task_list);

		tmp = tmp->next;

#if WAITQUEUE_DEBUG
		CHECK_MAGIC(curr->__magic);
#endif
		p = curr->task;
		state = p->state;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
		if (state & mode) {
#if WAITQUEUE_DEBUG
			curr->__waker = (long)__builtin_return_address(0);
#endif
			/*
			 * If waking up from an interrupt context then
			 * prefer processes which are affine to this
			 * CPU.
			 */
			if (irq && (curr->flags & wq_mode & WQ_FLAG_EXCLUSIVE)) {
				if (!best_exclusive)
					best_exclusive = p;
				if (p->processor == best_cpu) {
					best_exclusive = p;
					break;
				}
			} else {
				if (sync)
					wake_up_process_synchronous(p);
				else
					wake_up_process(p);
				if (curr->flags & wq_mode & WQ_FLAG_EXCLUSIVE)
					break;
			}
		}
	}
	if (best_exclusive) {
		if (sync)
			wake_up_process_synchronous(best_exclusive);
		else
			wake_up_process(best_exclusive);
	}
	wq_write_unlock_irqrestore(&q->lock, flags);
out:
	return;
}
```

可以看出，当一个进程正在等待进入一个临界区时，它所等待的是独占资源（在使用期间需要独占的资源）的释放。而进入了一个临界区的进程则占用了一项独占资源。如果一个进程进入了一个临界区 A，而又企图进入另外一个临界区 B 的话，那就可能会因为进入不了那个临界区，也就是得不到所需的资源，而只好在 B 的队列中等待。那么，所等待的资源又在谁的手里呢？如果已经占有了那项资源的进程恰好也正在 A 的队列中等待，那就发生了所谓的“死锁”，因为此时两个进程都无法向前推进而到达可以释放资源的那一步。显然，对共享资源（在使用期间也允许共享的资源）的使用是不会导致死锁的。在 Linux 系统中，多数的资源都是可共享的，而独占资源的使用则置于临界区中。只要保证不在一个临界区中企图进入另一个临界区，那就不会发生死锁，而这也是防止死锁的最简单的办法。进一步，即使在一个临界区中企图进入另一个临界区，但是如果为所有的临界区排好一个次序，

所有的进程在进入临界区时都遵守相同的次序（例如，只能先进 A 后进 B，而不允许先进 B 后进 A），则也不会发生上述因循环等待而引起的死锁。在目前的内核中尚无防止和化解死锁的措施，也没有防止进程在一个临界区中不按次序进入另一个临界区的措施。所以，这也是将来可加以改进的一个方面。

在内核中，需要“互斥”的不仅仅是进程与进程之间，干扰也可能发生于进程与中断服务程序（或 bh 函数）之间。同时，“信号量”也并非防止进程间，特别是在不同处理器上运行的进程之间互相干扰的惟一手段。例如，关中断无疑是保证同一处理器中进程与中断服务程序间互斥的一种手段，但是它不能防止来自另一处理器上的中断服务程序或进程的干扰。

另一种有效的手段就是加锁。读者在前面__down()的代码中看到的 spin_lock_irq()和 spin_unlock_irq()就是其中之一。特别是在多处理器 SMP 结构的系统中，由软件实现的各种锁尤其起着无可替代的作用。文件 include/1inux/spinlock.h 中定义了一些加锁操作：

==================== include/linux/spinlock.h 6 32 ====================

```
/*
 * These are the generic versions of the spinlocks and read-write
 * locks..
 */
#define spin_lock_irqsave(lock, flags)        do { local_irq_save(flags);      spin_lock(lock); } while (0)
#define spin_lock_irq(lock)           do { local_irq_disable();        spin_lock(lock); } while (0)
#define spin_lock_bh(lock)            do { local_bh_disable();         spin_lock(lock); } while (0)

#define read_lock_irqsave(lock, flags)        do { local_irq_save(flags);      read_lock(lock); } while (0)
#define read_lock_irq(lock)           do { local_irq_disable();        read_lock(lock); } while (0)
#define read_lock_bh(lock)            do { local_bh_disable();         read_lock(lock); } while (0)

#define write_lock_irqsave(lock, flags)       do { local_irq_save(flags);     write_lock(lock); } while (0)
#define write_lock_irq(lock)              do { local_irq_disable();       write_lock(lock); } while (0)
#define write_lock_bh(lock)           do { local_bh_disable();        write_lock(lock); } while (0)

#define spin_unlock_irqrestore(lock, flags)   do { spin_unlock(lock);  local_irq_restore(flags); } while (0)
#define spin_unlock_irq(lock)             do { spin_unlock(lock);  local_irq_enable();      } while (0)
#define spin_unlock_bh(lock)              do { spin_unlock(lock);  local_bh_enable();       } while (0)

#define read_unlock_irqrestore(lock, flags)   do { read_unlock(lock);  local_irq_restore(flags); } while (0)
#define read_unlock_irq(lock)             do { read_unlock(lock);  local_irq_enable();      } while (0)
#define read_unlock_bh(lock)              do { read_unlock(lock);  local_bh_enable();       } while (0)

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define write_unlock_irqrestore(lock, flags)  do { write_unlock(lock); local_irq_restore(flags); } while (0)
#define write_unlock_irq(lock)             do { write_unlock(lock); local_irq_enable();      } while (0)
#define write_unlock_bh(lock)              do { write_unlock(lock); local_bh_enable();       } while (0)
```

首先来看看同一组加锁操作之间的不同。例如，spin_lock_irqsave()与 spin_lock_irq()之间的区别仅在于前者调用 local_irq_save()而后者调用 local_irq_disable()。相应的解锁操作 spin_unlock_irqsave()与 spin_unlock_irq()之间的区别也就因此而不同，前者调用 local_irq_restore()而后者调用 local_irq_enable()。

再来看看不同组的加锁操作有什么不同。例如，spin_lock_irq()和 read_lock_irq()之间的区别仅在于前者调用 spin_lock()而后者调用 read_lock()。

每一个操作都包含了两部分。一部分是操作名以 local_开头的，其作用是关闭或开启本处理器上的中断响应。另一部分是操作名以_lock 结尾的，其作用是防止来自其它处理器的干扰。我们先来看处理开中断 / 关中断的操作，这都是在 include/asm-i386/system.h 中定义的：

==================== include/asm-i386/system.h 304 307 ====================

```
#define local_irq_save(x)   __asm__ __volatile__("pushfl ; popl %0 ; cli":"=g" (x): /* no input */ :"memory")
#define local_irq_restore(x)     __restore_flags(x)
#define local_irq_disable() __cli()
#define local_irq_enable()  __sti()
```

可见，local_irq_save()和 local_irq_disable()都通过 cli 指令来关闭中断，但是前者先把当前的处理器状态标志寄存器的内容保存起来，因为其中的 IF 标志就反映当前的中断是开着还是关着（指令 cli 就是把 IF 标志位清 0），以便在去锁时加以恢复。由于状态标志寄存器并非通用寄存器，所以要用 push 和 pop 指令经过堆栈将其内容保存到参数 x 中。相应地，local_irq_restore()与 local_irq_enable()的区别也在于此。

再来看 spin_lock()，其定义在 spinlock.h：

==================== include/asm-i386/spinlock.h 78 91 ====================

```
static inline void spin_lock(spinlock_t *lock)
{
#if SPINLOCK_DEBUG
	__label__ here;
here:
	if (lock->magic != SPINLOCK_MAGIC) {
printk("eip: %p\n", &&here);
		BUG();
	}
#endif
	__asm__ __volatile__(
		spin_lock_string
		:"=m" (lock->lock) : : "memory");
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

参数 lock 的类型为 spinlock_t，定义于 include/asm-i386/spinlock.h：

==================== include/asm-i386/spinlock.h 17 26 ====================

```
/*
 * Your basic SMP spinlocks, allowing only a single CPU anywhere
 */

typedef struct {
     volatile unsigned int lock;
#if SPINLOCK_DEBUG
     unsigned magic;
#endif
} spinlock_t;
```

如果不考虑调试，这实际上就是一个无符号整数，但是这样有利于防止 gcc 在编译过程加以有害的“优化”。代码中引用的 spin_lock_string 又是一个宏定义：

==================== include/asm-i386/spinlock.h 50 60 ====================

```
#define spin_lock_string \
     "\n1:\t" \
     "lock ; decb %0\n\t" \
     "js 2f\n" \
     ".section .text.lock,\"ax\"\n" \
     "2:\t" \
     "cmpb $0,%0\n\t" \
     "rep;nop\n\t" \
     "jle 2b\n\t" \
     "jmp 1b\n" \
     ".previous"
```

这里的%0 与参数 lock->lock 相结合。这里的指令 decb 将操作数，即 lock->lock 减 1，而后缀 b 则表示操作数为 8 位。这条指令带有前缀“ lock”，表示在执行时要将总线锁住，不让其它处理器访问，以此来保证该条指令执行的“原子性”。减 1 以后，要是结果非负（符号位为 0）则加锁成功，所以就返回了。如果发现减 1 以后的结果成了负数，那就表示已经有其它操作先加了锁，因此被锁在了门外，这时就转移到标号“ 2”处循环测试，等待加锁者去锁后将 lock->lock 设置成大于 0，然后又试着加锁。

从代码中可以看出，如果 lock->lock 的值原来就已经是 0 或负数，则处理器不断地循环测试它的值，直至其变成大于 0 为止，所以才有 spin_lock 这个名字。所谓 spin 就是“连轴转”的意思。处理器不断地这么连轴转，当然是在做无用功。那么为什么不像在对信号量的 down()操作那样进入睡眠，把 CPU 让给其它进程来做些有用功呢？这是因为想要加锁的这段程序未必是在一个进程的上下文中调用的，它可能调用自一段中断服务程序或者 bh 函数，根本就不是可调度的。这也说明，加锁的时间不能太长，否则就可能太浪费了。

至于 spin_unlock()的代码那就很简单了：

==================== include/asm-i386/spinlock.h 93 104 ====================

```
static inline void spin_unlock(spinlock_t *lock)
{
```

```
#if SPINLOCK_DEBUG
	if (lock->magic != SPINLOCK_MAGIC)
		BUG();
	if (!spin_is_locked(lock))
		BUG();
#endif
	__asm__ __volatile__(
		spin_unlock_string
		:"=m" (lock->lock) : : "memory");
}
```

同样，spin_unlock_string 也是个宏定义：

==================== include/asm-i386/spinlock.h 62 66 ====================

```
/*
 * This works. Despite all the confusion.
 */
#define spin_unlock_string \
	"movb $1,%0"
```

代码中的指令 movb 将 lock->lock 设置成 1，如此而已。这条指令不带有前缀 “ lock”，因为指令 movb 的操作本身就是原子性的。相比之下，前面的指令 decb 因为涉及 “读—改—写” 周期，所以从总线角度看不是原子性的。

读者也许会问，既然被锁在门外的处理器只是在做无用功，那为何不干脆就把总线锁了，一直到要做的事情完成以后才开放呢？这还是不同的，正在连轴转做无用功的处理器仍能响应中断。而如果干脆把总线锁了那就连中断也不能响应了。再说，系统中也还可能有其它处理器，只要不想进入同一段代码或受同一把锁保护的代码，就可以继续运行。

再来看 read_lock()和 write_lock()，其实现也是大同小异，我们把这些代码（也在 spinlock.h 中）留给读者自己阅读：

==================== include/asm-i386/spinlock.h 135 165 ====================

```
/*
 * On x86, we implement read-write locks as a 32-bit counter
 * with the high bit (sign) being the "contended" bit.
 *
 * The inline assembly is non-obvious. Think about it.
 *
 * Changed to use the same technique as rw semaphores.  See
 * semaphore.h for details.  -ben
 */
/* the spinlock helpers are in arch/i386/kernel/semaphore.c */

static inline void read_lock(rwlock_t *rw)
{
#if SPINLOCK_DEBUG
	if (rw->magic != RWLOCK_MAGIC)
		BUG();
```

```
#endif
	__build_read_lock(rw, "__read_lock_failed");
}

static inline void write_lock(rwlock_t *rw)
{
#if SPINLOCK_DEBUG
	if (rw->magic != RWLOCK_MAGIC)
		BUG();
#endif
	__build_write_lock(rw, "__write_lock_failed");
}

#define read_unlock(rw)	asm volatile("lock ; incl %0" :"=m" ((rw)->lock) : : "memory")
#define write_unlock(rw)	asm volatile("lock ; addl $" RW_LOCK_BIAS_STR ",%0":"=m" ((rw)->lock) : : "memory")
```

代码中引用的一些宏操作和宏定义为：

==================== include/asm-i386/rwlock.h 20 81 ====================

```
#define RW_LOCK_BIAS		 0x01000000
#define RW_LOCK_BIAS_STR	"0x01000000"

#define __build_read_lock_ptr(rw, helper)   \
	asm volatile(LOCK "subl $1,(%0)\n\t" \
		     "js 2f\n" \
		     "1:\n" \
		     ".section .text.lock,\"ax\"\n" \
		     "2:\tcall " helper "\n\t" \
		     "jmp 1b\n" \
		     ".previous" \
		     ::"a" (rw) : "memory")

#define __build_read_lock_const(rw, helper)   \
	asm volatile(LOCK "subl $1,%0\n\t" \
		     "js 2f\n" \
		     "1:\n" \
		     ".section .text.lock,\"ax\"\n" \
		     "2:\tpushl %%eax\n\t" \
		     "leal %0,%%eax\n\t" \
		     "call " helper "\n\t" \
		     "popl %%eax\n\t" \
		     "jmp 1b\n" \
		     ".previous" \
		     :"=m" (*(volatile int *)rw) : : "memory")

#define __build_read_lock(rw, helper)do { \
				if (__builtin_constant_p(rw)) \
					__build_read_lock_const(rw, helper); \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
49                              else \
50                                   __build_read_lock_ptr(rw, helper); \
51                         } while (0)
52
53   #define __build_write_lock_ptr(rw, helper) \
54        asm volatile(LOCK "subl $" RW_LOCK_BIAS_STR ",(%0)\n\t" \
55                 "jnz 2f\n" \
56                 "1:\n" \
57                 ".section .text.lock,\"ax\"\n" \
58                 "2:\tcall " helper "\n\t" \
59                 "jmp 1b\n" \
60                 ".previous" \
61                 ::"a" (rw) : "memory")
62
63   #define __build_write_lock_const(rw, helper) \
64        asm volatile(LOCK "subl $" RW_LOCK_BIAS_STR ",(%0)\n\t" \
65                 "jnz 2f\n" \
66                 "1:\n" \
67                 ".section .text.lock,\"ax\"\n" \
68                 "2:\tpushl %%eax\n\t" \
69                 "leal %0,%%eax\n\t" \
70                 "call " helper "\n\t" \
71                 "popl %%eax\n\t" \
72                 "jmp 1b\n" \
73                 ".previous" \
74                 :"=m" (*(volatile int *)rw) : : "memory")
75
76   #define __build_write_lock(rw, helper)    do { \
77                              if (__builtin_constant_p(rw)) \
78                                   __build_write_lock_const(rw, helper); \
79                              else \
80                                   __build_write_lock_ptr(rw, helper); \
81                         } while (0)
```

调用__build_read_lock()和__build_write_lock()时的第二个参数都是函数指针，分别为__read_lock_failed 和__write_lock_failed（见 152 行和 161 行），其代码如下：

==================== arch/i386/kernel/semaphore.c 426 453 ====================

```
426  #if defined(CONFIG_SMP)
427  asm(
428  "
429  .align   4
430  .globl   __write_lock_failed
431  __write_lock_failed:
432       " LOCK "addl  $" RW_LOCK_BIAS_STR ",(%eax)
433  1:   cmpl $" RW_LOCK_BIAS_STR ",(%eax)
434       jne  1b
435
436       " LOCK "subl  $" RW_LOCK_BIAS_STR ",(%eax)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         jnz   __write_lock_failed
         ret


    .align    4
    .globl    __read_lock_failed
    __read_lock_failed:
         lock ; incl    (%eax)
    1:   cmpl $1,(%eax)
         js    1b

         lock ; decl    (%eax)
         js    __read_lock_failed
         ret
    "
    );
    #endif
```

值得指出的是，如果 CPU 进入了一段加锁的代码 A 以后又企图进入另一段加锁的代码 B，那就有可能在那里被关在门外，而如果已经在 B 中的处理器恰好又因企图进入 A 而也被关在了门外，因此两个处理器都陷入了连轴转，那就形成了死锁。这跟在信号量上通过 down()操作嵌套地进入了临界区时可能会形成的死锁是一致的。防止此种死锁的手段主要有：

(1) 不允许在进入加锁的代码以后再进入其它加锁的代码，这是最简单的。

(2) 如果允许这样做的话，就要为所有加锁的代码段建立一个统一的次序，例如必须是先进入 A 后进入 B。

目前的 Linux 内核中尚未采取措施来防止这种死锁，这也有待于将来进一步的改进。不过，并非每个程序员都需要编写在内核中运行的程序，负责开发内核程序（如设备驱动）的程序员通常总是比较有经验、水平比较高的人，他们自会注意这个问题。另一方面，Linux 系统中的多数资源都是可共享的，需要放在临界区中或者加锁的代码是很少、很短的。所以，在实际使用中死锁并不是一个很现实的问题。

还有，临界区和加锁在概念上是类似的，但对系统影响的程度却不同，所以在使用时要加以区分。如果不问青红皂白，动不动就加锁，那就好像为了抓小偷而全城戒严一样，属于“防范过当”了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第5章 文件系统

5.1 概述

若要问构成一个“操作系统”的最重要的部件是什么，那就莫过于进程管理和文件系统了。事实上，有些操作系统（如一些“嵌入式”系统）可能有进程管理而没有文件系统；而另一些操作系统（如MSDOS）则有文件系统而没有进程管理。可是，要是二者都没有，那就称不上“操作系统”了。在本书的前几章中已经讲述了与 Linux 的进程管理有关的内容，从现在开始，让我们把注意力转向它的文件系统。

“文件系统”这个词的含义比较模糊。首先，其中“文件”的含义就有狭义与广义之分。狭义地说，“文件”是指“磁盘文件”，进而可以是有组织有次序地存储于任何介质（包括内存）中的一组信息。广义地说，则 Unix 从一开始时就把外部设备都当成“文件”。从这个意义上讲，凡是可以产生或消耗信息的都是文件。以在网络环境中用来收发报文的“插口”机制来说，它就并不代表存储着的信息，但是插口的发送端“消耗”信息，而接收端则“产出”信息，所以把插口看成文件是合乎逻辑的。可是，即使抛开“文件”这个词的模糊性不说，“文件系统”这个词又进一步有几种不同的含义，要根据上下文才能加以区分：

(1) 指一种特定的文件格式。例如，我们说 Linux 的文件系统是 Ext2，MSDOS 的文件系统是 FAT16，而 Windows NT 的文件系统是 NTFS 或 FAT32，就是指这个意思。

(2) 指按特定格式进行了“格式化”的一块存储介质。当我们说“安装”或“拆卸”一个文件系统时，指的就是这个意思。

(3) 指操作系统中（通常在内核中）用来管理文件系统以及对文件进行操作的机制及其实现，这就是本章的主要话题。

Linux 最初采用的是 minix 的文件系统，但是 minix 只是一种实验性（用于教学）的操作系统，其文件系统的大小仅限于 64M 字节，文件名长度限于 14 个字符。所以，经过一段时间的改进和发展，特别是吸取了多年来对传统 Unix 文件系统的各种改进所累积起的经验，最后形成了现在的 Ext2 文件系统。这个文件系统可以说就是“ Linux 文件系统”。

除 Linux 本身的文件系统 Ext2 外，设计人员很早就注意到了如何使 Linux 支持其它各种不同文件系统的问题。要实现这个目的，就要将对各种不同文件系统的操作和管理纳入到一个统一的框架中。让内核中的文件系统界面成为一条文件系统“总线”，使得用户程序可以通过同一个文件系统操作界面，也就是同一组系统调用，对各种不同的文件系统（以及文件）进行操作。这样，就可以对用户程序隐去各种不同文件系统的实现细节，为用户程序提供一个统一的、抽象的、虚拟的文件系统界面，这就是所谓“虚拟文件系统 ” VFS（Virtual Filesystem System）。这个抽象的界面主要由一组标准的、抽象的文件操作构成，以系统调用的形式提供于用户程序，如 read()、write()、lseek()等等。这样，用户程序就可以把所有的文件都看作一致的、抽象的“ VFS 文件”，通过这些系统调用对文件进行操作，而无

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

需关心具体的文件属于什么文件系统以及具体文件系统的设计和实现。例如，在 Linux 操作系统中，可以将 DOS 格式的磁盘或分区（即文件系统）“安装”到系统中，然后用户程序就可以按完全相同的方式访问这些文件，就好像它们也是 Ext2 格式的文件一样。

如果把内核比拟为 PC 机中的“母板”，把 VFS 比拟为“母板”上的一个“插槽”，那么每个具体的文件系统就好像一块“接口卡”。不同的接口卡上有不同的电子线路，但是它们与插槽的连接有几条线、每条线干什么用则是有明确定义的。同样，不同的文件系统通过不同的程序来实现其各种功能，但是与 VFS 之间的界面则是有明确定义的。这个界面的主体就是一个 file_operations 数据结构，其定义在 include/linux/fs.h 中：

==================== include/linux/fs.h 768 790 ====================

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev can be called
 *   without the big kernel lock held in all filesystems.
 */
struct file_operations {
	struct module *owner;
	loff_t (*llseek) (struct file *, loff_t, int);
	ssize_t (*read) (struct file *, char *, size_t, loff_t *);
	ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
	int (*readdir) (struct file *, void *, filldir_t);
	unsigned int (*poll) (struct file *, struct poll_table_struct *);
	int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
	int (*mmap) (struct file *, struct vm_area_struct *);
	int (*open) (struct inode *, struct file *);
	int (*flush) (struct file *);
	int (*release) (struct inode *, struct file *);
	int (*fsync) (struct file *, struct dentry *, int datasync);
	int (*fasync) (int, struct file *, int);
	int (*lock) (struct file *, int, struct file_lock *);
	ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
	ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
};
```

每种文件系统都有自己的 file_operations 数据结构，结构中的成分几乎全是函数指针，所以实际上是个函数跳转表，例如 read 就指向具体文件系统用来实现读文件操作的入口函数。如果具体的文件系统不支持某种操作，其 file_operations 结构中的相应函数指针就是 NULL。我们不在这里介绍这个结构中各种指针的用途，以后读者将会看到其中主要的一些操作的典型代码。

每个进程通过“打开文件”（open()）与具体的文件建立起连接，或者说建立起一个读写的“上下文”。这种连接以一个 file 数据结构作为代表，结构中有个 file_operations 结构指针 f_op。将 file 结构中的指针 f_op 设置成指向某个具体的 file_operations 结构，就指定了这个文件所属的文件系统，并且与具体文件系统所提供的一组函数挂上了钩，就好像把具体的“接口卡”插入到了“插槽”中。读者以后会看到有关的详情。

Linux 内核中对 VFS 与具体文件系统的关系划分可以用图 5.1 表示。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

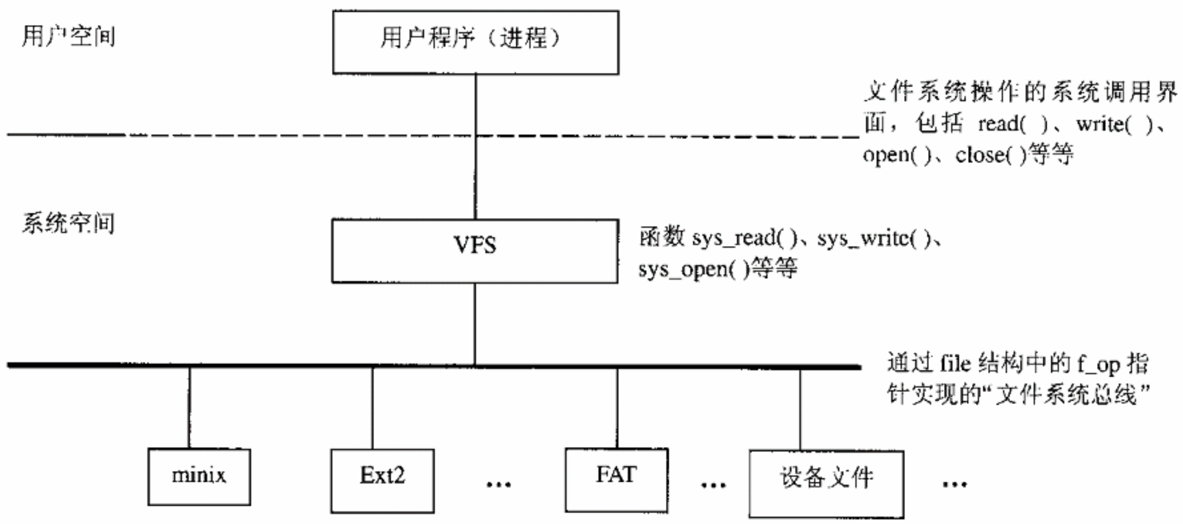


图 5.1　VFS 与具体文件系统的关系示意图

进程与文件的连接，即“已打开文件”，是进程的一项“财产”，归具体的进程所有。代表着这种连接的 file 结构必然与代表着进程的 task_struct 数据结构存在着联系。在“进程与进程调度”一章中我们看过这个数据结构的定义，但是那时候忽略了与文件系统有关的内容。现在把 task_struct 结构中与此有关的几行再列出如下（include/linux/sched.h）：

==================== include/linux/sched.h 277 277 ====================
```
struct task_struct {
    . . . . . .
```
==================== include/linux/sched.h 375 378 ====================
```
/* filesystem information */
     struct fs_struct *fs;
/* open file information */
     struct files_struct *files;
    . . . . . .
```
==================== include/linux/sched.h 397 397 ====================
```
};
```

这里有两个指针 fs 和 files，一个指向 fs_struct 数据结构，是关于文件系统的信息；另一个指向 files_struct 数据结构，是关于已打开文件的信息。先看 fs_struct 结构，它的定义在 include/linux/fs_struct.h 中：

==================== include/linux/fs_struct.h 5 11 ====================
```
struct fs_struct {
     atomic_t count;
     rwlock_t lock;
     int umask;
     struct dentry * root, * pwd, * altroot;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
```

结构中有六个指针。前三个是 dentry 结构指针，就是 root、pwd 以及 altroot。这些指针各自指向代表着一个“目录项”的 dentry 数据结构，里面记录着文件的各项属性，如文件名、访问权限等等。其中 pwd 则指向进程当前所在的目录；而 root 所指向的 dentry 结构代表着本进程的“根目录”，那就是当用户登录进入系统时所“看到”的根目录；至于 altroot 则为用户设置的“替换根目录”，我们以后还会讲到。实际运行时这三个目录不一定都在同一个文件系统中，例如进程的根目录通常是安装于“/”节点上的 Ext2 文件系统中，而当前工作目录则可能安装于/dosc 的一个 DOS 文件系统中。在文件系统的操作中这些安装点起着重要的作用而常常要用到，所以后三个指针就各自指向代表着这些“安装”的 vfsmount 数据结构。注意，fs_struct 结构中的信息都是与文件系统和进程有关的，带有全局性的（对具体的进程而言），而与具体的已打开文件没有什么关系。例如进程的根目录在 Ext2 文件系统中，当前工作目录在 DOS 文件系统中，而一个具体的已打开文件却可能是设备文件。

与具体已打开文件有关的信息在 file 结构中，而 files_struct 结构的主体就是一个 file 结构数组。每打开一个文件以后，进程就通过一个“打开文件号”fid 来访问这个文件，而 fid 实际上就是相应 file 结构在数组中的下标。如前所述，每个 file 结构中有个指针 f_op，指向该文件所属文件系统的 file_operations 数据结构。同时，file 结构中还有个指针 f_dentry，指向该文件的 dentry 数据结构。那么，为什么不干脆把文件的 dentry 结构放在 file 结构里面，而只是让 file 结构通过指针来指向它呢？这是因为一个文件只有一个 dentry 数据结构，而可能有多个进程打开它，甚至同一个进程也可能多次打开它而建立起多个读写上下文。同理，每种文件系统只有一个 file_operations 数据结构，它既不专属于某个特定的文件，更不专属于某个特定的上下文。

每个文件除有一个“目录项”即 dentry 数据结构以外，还有一个“索引节点”即 inode 数据结构，里面记录着文件在存储介质上的位置与分布等信息。同时，dentry 结构中有个 inode 结构指针 d_inode 指向相应的 inode 结构。读者也许要问，既然一个文件的 dentry 结构和 inode 结构都在从不同的角度描述这个文件各方面的属性，那为什么不“合二为一”，而要“一分为二”呢？其实，dentry 结构与 inode 结构所描述的目标是不同的，因为一个文件可能有好几个文件名，而通过不同的文件名访问同一个文件时的权限也可能不同。所以，dentry 结构所代表的是逻辑意义上的文件，记录的是其逻辑上的属性。而 inode 结构所代表的是物理意义上的文件，记录的是其物理上的属性；它们之间的关系是多对一的关系。

前面我们说虚拟文件系统 VFS 与具体的文件系统之间的界面的“主体”是 file_operations 数据结构，是因为除此之外还有一些其它的数据结构。其中主要的还有与目录项相联系的 dentry_operations 数据结构和与索引节点相联系的 inode_operations 数据结构。这两个数据结构中的内容也都是一些函数指针，但是这些函数大多只是在打开文件的过程中使用，或者仅在文件操作的“底层”使用（如分配空间），所以不像 file_operations 结构中那些函数那么常用，或者不那么有“知名度”。以 dentry_operations 为例，其定义在 include/linux/dcache.h 中：

==================== include/linux/dcache.h 79 86 ====================

```
struct dentry_operations {
        int (*d_revalidate)(struct dentry *, int);
        int (*d_hash) (struct dentry *, struct qstr *);
        int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        int (*d_delete)(struct dentry *);
        void (*d_release)(struct dentry *);
        void (*d_iput)(struct dentry *, struct inode *);
};
```

这里 d_delete 是指向具体文件系统的“删除文件”操作的入口函数，d_release 则用于“关闭文件”操作。还有个函数指针 d_compare 很有意思，它用于文件名的比对。读者可能感到奇怪，文件名的比对就是字符串的比对，这难道也是因文件系统而异？其实不足为奇，想一想：有的文件系统中文件名的长度限于 8 个字符，有的则可以有 255 个字符；有的文件系统容许在文件名里有空格，有的则不容许；有的支持汉字，有的则不支持。所以，文件名的比对确实因文件系统而异。

总之，具体文件系统与虚拟文件系统 VFS 间的界面是一组数据结构，包括 file_operations、dentry_operations、inode_operations，还有其它（此处暂从略）。原则上每种文件系统都必须在内核中提供这些数据结构，后面我们还要深入讨论。

虽然我们尚未深入地讲述文件系统的内部结构和操作，读者可以从图 5.2 看到文件系统内部结构的基本情况。我们姑且称之为“文件系统逻辑结构图”，以后读者将需要反复地回过来看这幅结构示意图。至于这幅图中各个数据结构的分配与设置，也就是这幅图的构筑，则会在“打开文件”一节中叙述。

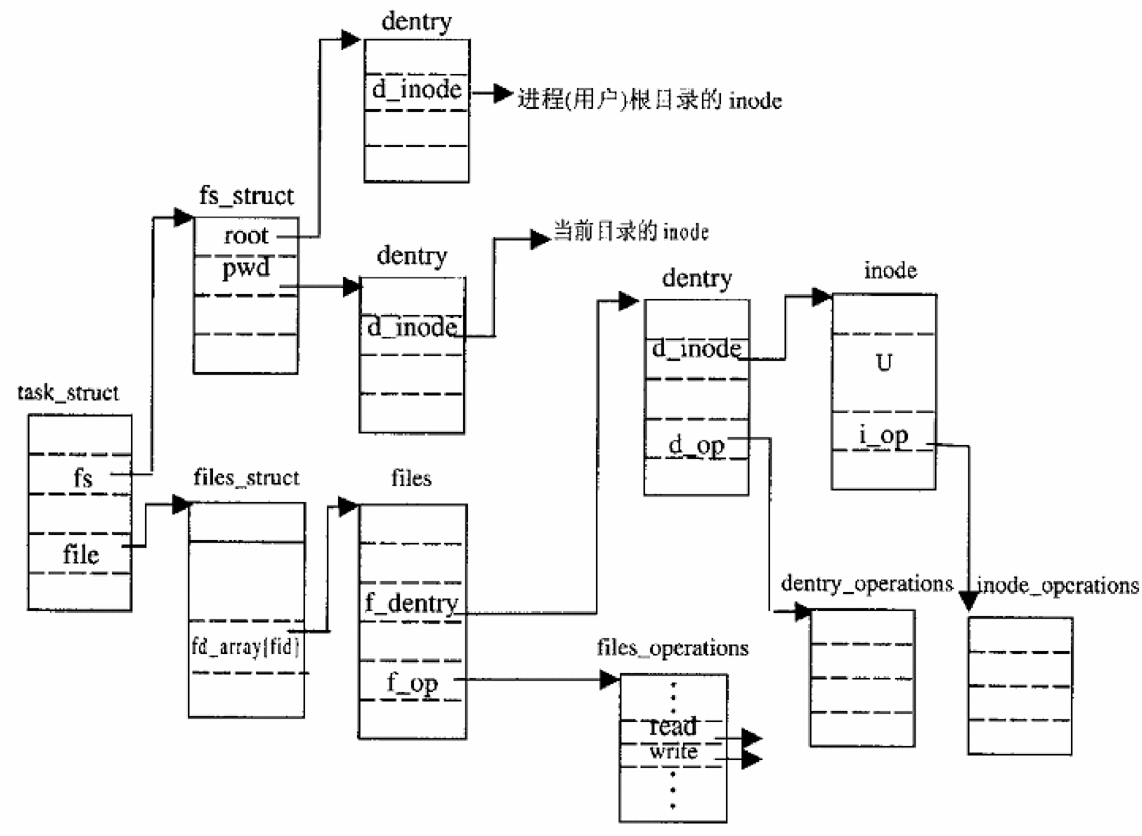


图 5.2　Linux 文件系统逻辑结构图

PDF-XCHANGE Click to buy NOW! www.docu-track.com

那么，Linux 到底支持哪一些具体的文件系统呢？数据结构 inode 中有一个成分 u，是一个 union。根据具体文件系统的不同，可以将这个 union 解释成不同的数据结构。例如，当 inode 所代表的文件是个插口（socket）时，u 就用作 socket 数据结构；当 inode 所代表的文件属于 Ext2 文件系统时，u 就用作 Ext2 文件系统的详细描述结构 ext2_inode_info。所以，看一下对这个 union 的定义就可以看出 Linux 目前支持多少种文件系统，这个定义是 inode 数据结构定义的一部分，在文件 include/linux/fs.h 中：

==================== include/linux/fs.h 433 460 ====================

```
	union {
		struct minix_inode_info		minix_i;
		struct ext2_inode_info		ext2_i;
		struct hpfs_inode_info		hpfs_i;
		struct ntfs_inode_info		ntfs_i;
		struct msdos_inode_info		msdos_i;
		struct umsdos_inode_info	umsdos_i;
		struct iso_inode_info		isofs_i;
		struct nfs_inode_info		nfs_i;
		struct sysv_inode_info		sysv_i;
		struct affs_inode_info		affs_i;
		struct ufs_inode_info		ufs_i;
		struct efs_inode_info		efs_i;
		struct romfs_inode_info		romfs_i;
		struct shmem_inode_info		shmem_i;
		struct coda_inode_info		coda_i;
		struct smb_inode_info		smbfs_i;
		struct hfs_inode_info		hfs_i;
		struct adfs_inode_info		adfs_i;
		struct qnx4_inode_info		qnx4_i;
		struct bfs_inode_info		bfs_i;
		struct udf_inode_info		udf_i;
		struct ncp_inode_info		ncpfs_i;
		struct proc_inode_info		proc_i;
		struct socket			socket_i;
		struct usbdev_inode_info	usbdev_i;
		void				*generic_ip;
	} u;
```

其中有些成分是不言自明的，如 ext2、msdos 等，但是多数都需要一点简短的说明：

hpfs——IBM 为 PC 开发的 OS/2 操作系统所采用的文件系统。这种格式只用于硬盘，而 OS/2 所用的软盘则与 msdos 相同。

ntfs——Windows NT 的文件系统。

umsdos——一种特殊的“文件系统”，用 msdos 文件系统来模拟 Ext2 文件系统。其好处是可以在磁盘上的 DOS 分区直接运行 Linux，而不需要先重新区并格式化。坏处当然也不少，首先是降低了运行的速度，而且这样一来就对 DOS 文件系统的病毒失去了“免疫力”。

isofs——用于 CDROM（光盘）。

nfs——“网络文件系统”NFS。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

sysv——Unix 系统 V 的文件系统 S5FS。

affs——BSD 对 S5FS 作了很大的改进，改进后的文件系统称为“快速文件系统 ” FFS。由于当时 Amiga 公司在其操作系统 AmigaOS 中采用了这种文件系统，所以称为 affs。

ufs——这是 FFS 的另一种实现，广泛适用于 BSD 的各种版本以及各种 Unix 变种（如 SunOS、Solaris、FreeBSD、NetBSD、OpenBSD 以及 Nextstep 等等）版本，因而实际上成为了“ Unix File System”，所以称为 ufs。

efs——Silicon Graphics 的 IRIX 文件系统。

romfs——“ 只读”文件系统。顾名思义，这种文件系统可以建立在只读介质上，如 EPROM、PROM 等。还有一个特点是这种文件系统的实现在内核中所占的“地盘”很小。例如，msdos 文件系统在内核中约占 30K 字节，nfs 文件系统约占 57K 字节，而 romfs 只占一个页面。所以，romfs 常常比较适合于一些“嵌入”式系统。

coda——也是一种网络文件系统，是对 nfs 的一种改进。

smbfs——即 samba，使 Win 95、Win NT 等系统可以通过网络访问 Linux 文件系统。

hfs——Apple Macintosh 的文件系统。

adfs——Acorn 公司开发了一种基于 ARM 处理器的 RISC PC，其操作系统称为 RISCOS，而文件系统即为 “ Acorn Disk Filing System”。

qnx4——QNX 是一个操作系统，常用于“嵌入式”系统，其文件系统为 QNXFS。

bfs——用于 SCO Unix Ware V 的一种文件系统

udf——“ Universal Disk File”最新的“通用文件系统”，既用于 DVD 和可写光盘，也可用于硬盘。

ncpfs——Novell NetWare 文件系统。

proc——目录/proc 下的特殊文件，这些文件对系统管理和程序调试都很有用处。

usbdev——“ 通用串行总线 ” USB 的驱动程序。

这个 union 的定义只是大致地反映了 Linux 内核目前所支持的各种文件系统，因为这是以 inode 结构中这一部分空间的不同用法和解释为基础的。如果两种文件系统对这个 union 的解释相同，那就不能从这个定义中反映出来了。

举例来说，早期 Linux 曾开发和使用了另一个文件系统 ext（还有一个文件系统叫 Xiafs），后来才发展到 Ext2(表示 ext 第 2 版)，现在的 Linux 内核也还支持这个文件系统(读者可以用命令“ man mount ”或 “ man fs”察看)，但是由于它在 inode 结构中对 u 的解释与 Ext2 相同，这里就看不出来了。另一方面，虽然原则上每个文件系统都有其自身的函数跳转表，即 file_operations 数据结构，但是反过来说，每个 file_operations 结构都代表着一个不同的文件系统就不确切了。读者以后在第 6 章的“管道”一节中可以看到，就在 Ext2 文件系统的框架中，光是用作管道的文件就根据读、写权限的不同而有一个不同的 file_operations 结构。

还有，在 Linux 系统中外部设备是视同文件的，所以从概念上讲每种不同的外部设备就相当于一种不同的文件系统。可是，在这个 union 的定义中却只列出了 usbdev 作为一种独立的文件系统，那么“块设备”又怎样？“字符设备”又怎样？“网络设备”又怎样？为什么这里都没有呢？原因就在于这些设备都不要求将 inode 结构中的这个 union 作不同的解释。同样的道理，对“特殊文件”，这里只列出了 proc 与 socket，但是用来实现“命名管道”的另一种特殊文件 FIFO 就没有在这里单独地列出。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

所以，inode 结构中的这个 union 反映了各种文件系统在部分数据结构上的不同，而 file_operations 结构则反应了它们在算法（操作）上的不同。

当一个文件系统代表着磁盘（或其它介质）上按特定格式组织的文件时，对每个文件的操作最终都要转化为对某一部分磁盘介质的操作，所以从层次的观点来看，在“文件系统”以下还有一层“设备驱动”。可是，既然设备（实际上是设备驱动）也是视作文件的，那么作为与文件“平级”的磁盘设备“文件”与作为文件系统“底层”的磁盘“设备”又有什么区别呢？这实际上反映了对磁盘介质上的数据的两种不同观点，一种是把它看成有结构、有组织的数据，而另一种则把它看成线性空间的数据。下列示意图（图 5.3）表示了不同文件的这种层次结构。

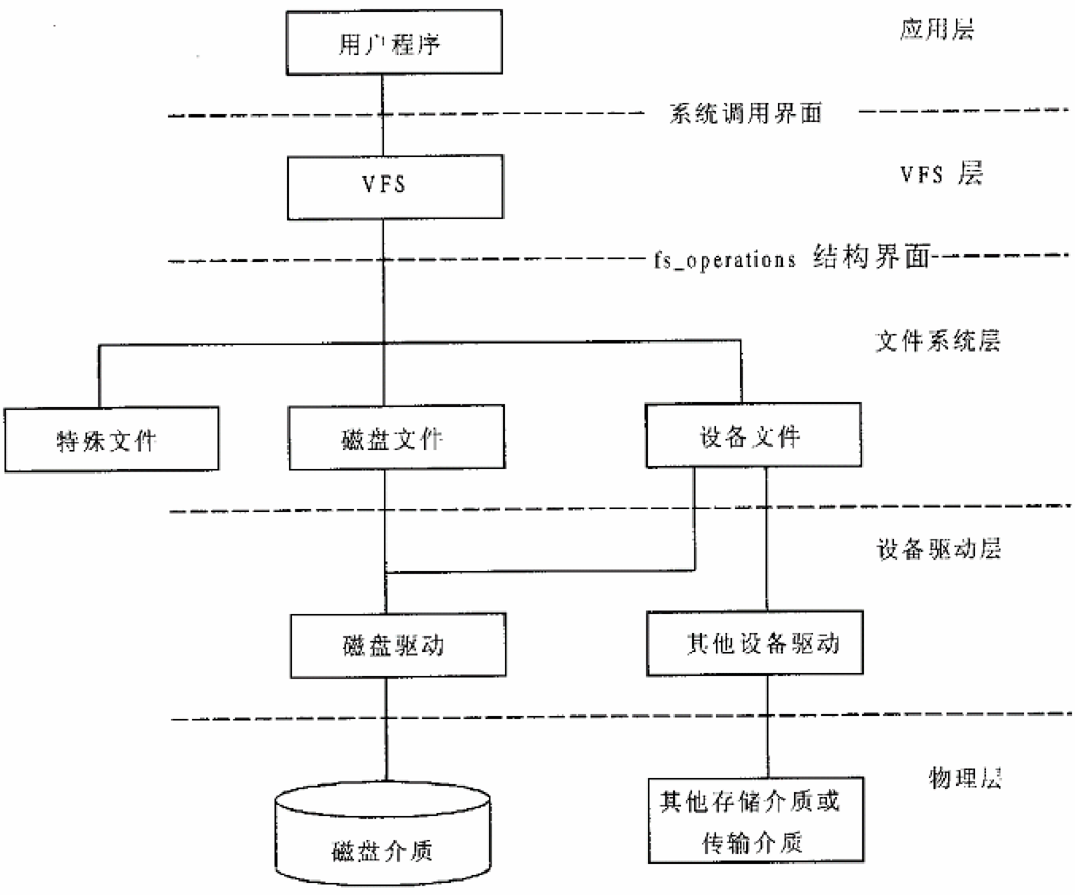


图 5.3　Linux 文件系统的层次结构

图中有三种不同类型的文件。

5.1.1 磁盘文件

磁盘文件也许应该称为“存储文件”，这就是狭义的、本来意义上的“文件”，通常以磁盘为存储

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

介质，但也可能采用其它介质。例如，romfs 就是采用 EPROM 之类的介质，而 RAMDISK 则在内存中模拟磁盘介质。所谓“文件”，就是按一定的组织形式存储在介质上的信息，所以一个“文件”其实包含了两方面的信息，一是存储的数据本身，还有一部分就是有关该文件的组织和管理的信息。对于磁盘文件来说，这两种信息必定全都存储在“文件系统”中，也就是磁盘上。其中与组织和管理有关的信息主要存储在文件的“索引节点”和“目录项”中。磁盘上的索引节点与前述（存在于内存中）的 inode 数据结构相似，但有所不同，可以看成是 inode 结构的简化了的版本。不过，磁盘上的索引节点不像 inode 结构那样会因断电而“挥发”，也不占用内存空间。每个文件都有一个并且只有一个索引节点。即使文件中暂时没有数据，其索引节点总是存在的。这类文件是本章的重点，不过我们在本书中将只关心 Ext2 文件系统。磁盘上的目录项也比内存中的 dentry 结构简单，存在于所谓“目录节点”中。目录节点实际上是一种特殊形式和用途的文件。

5.1.2 设备文件

设备文件同样包含有用于组织和管理的信息，同样有存储介质上的索引节点和目录项，但是却不一定有存储着的数据。根据设备类型和性质的不同，它可以是用于存储 / 读出的（如磁盘），也可以是用于接收 / 发送的（如网络卡），还可以是供采集 / 控制的（如一些机电设备），甚至可以是数种类型的结合。实际上，不管什么设备，在操作的过程中总要伴随着一定程度的数据采集和控制，通常都通过设备接口上的一个“控制 / 状态寄存器”进行，具体可参看本书下册“设备驱动”一章。

5.1.3 特殊文件

特殊文件在内存中也有 inode 数据结构和 dentry 数据结构，但是不一定在存储介质上有索引节点和目录项。与前两种文件主要的不同是：特殊文件一般都与外部设备无关，所涉及的介质通常就是内存以及 CPU 本身。当从一特殊文件“读”时，所读出的数据都是由系统内部按一定的规则临时生成出来的，或者从内存中收集、加工出来的，反之亦然。从这个意义上说，文件“ dev/null”就是一个特殊文件，凡是写入这个文件的数据全部都被丢弃了，根本就与外部设备无关。所以这个文件虽然在“ /dev ”目录下，实质上却是一个特殊文件。读者在“进程间通信”一章中将会看到用来实现“管道”的文件，特别是“命名管道”的 FIFO 文件，还有 Unix 域的 socket，也都属于特殊文件。在本章中我们还要介绍另一种重要的特殊文件，那就是在“ /proc”目录下的一系列文件。

三种不同类型的文件有一个共同点，那就是它们都有一些关于组织和管理的信息。因此，每个文件都有一个 inode。所谓 inode，也就是“索引节点”（或称“ i 节点”）的意思。要“访问”一个文件时，一定要通过它的索引才能知道这个文件是什么类型的文件（例如，是否设备文件）、是怎样组织的、文件中存储着多少数据、这些数据在什么地方以及其下层的驱动程序在哪儿等必要的信息。数据结构 inode 的定义在文件 include/linux/fs.h 中给出，我们把它列在这里，但是现在还不是有系统地解释结构中各个成分的时候，读者以后还得反复回过来看它的定义。最后，读者自己就能作出这种解释了。此处我们只对结构中的某些成分作一些介绍。先看 inode 的定义（include/linux/fs.h）：

==================== include/linux/fs.h 387 435 ====================

```
struct inode {
    struct list_head    i_hash;
    struct list_head    i_list;
    struct list_head    i_dentry;

    struct list_head    i_dirty_buffers;

    unsigned long       i_ino;
    atomic_t        i_count;
    kdev_t              i_dev;
    umode_t             i_mode;
    nlink_t             i_nlink;
    uid_t               i_uid;
    gid_t               i_gid;
    kdev_t              i_rdev;
    loff_t              i_size;
    time_t              i_atime;
    time_t              i_mtime;
    time_t              i_ctime;
    unsigned long       i_blksize;
    unsigned long       i_blocks;
    unsigned long       i_version;
    struct semaphore    i_sem;
    struct semaphore    i_zombie;
    struct inode_operations *i_op;
    struct file_operations  *i_fop;   /* former ->i_op->default_file_ops */
    struct super_block  *i_sb;
    wait_queue_head_t   i_wait;
    struct file_lock    *i_flock;
    struct address_space    *i_mapping;
    struct address_space    i_data;
    struct dquot        *i_dquot[MAXQUOTAS];
    struct pipe_inode_info  *i_pipe;
    struct block_device     *i_bdev;

    unsigned long       i_dnotify_mask; /* Directory notify events */
    struct dnotify_struct   *i_dnotify; /* for directory notifications */

    unsigned long       i_state;

    unsigned int        i_flags;
    unsigned char       i_sock;

    atomic_t        i_writecount;
    unsigned int        i_attr_flags;
    __u32               i_generation;
    union {
        struct minix_inode_info     minix_i;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        struct ext2_inode_info      ext2_i;
    . . . . . .
```

==================== include/linux/fs.h 460 461 ====================

```
    } u;
};
```

每个 inode 都有一个“i 节点号”i_ino，在同一文件系统中每个 i 节点号都是惟一的，内核中有时候会根据 i 节点号的杂凑值寻找其 inode 结构。同时，每个文件都有个“文件主”，最初是创建了这个文件的用户，但是可以改变。系统的每个用户都有一个用户号，即 uid，并且都属于某一个用户“组”，所以又有个组号 gid。因此，在 inode 结构中就相应地有 i_uid 和 i_gid 两个成分，以指明文件主的身分。

值得注意的是，inode 结构中有两个设备号，即 i_dev 和 i_rdev。首先，除特殊文件外，一个索引节点总得存储在某个设备上，这就是 i_dev。其次，如果索引节点所代表的并不是常规文件，而是某个设备，那就还要有个设备号，那就是 i_rdev。设备号实际上由两部分构成，即“主设备号”与“次设备号”。主设备号表示设备的种类，例如磁盘就分成软盘、IDE 硬盘、SCSI 硬盘等等。次设备号则表示系统内配备的同一种设备中的某个具体设备。

每当一个文件受到访问时，系统都要在这个文件的 inode 中留下时间印记，inode 结构中的 i_atime、i_mtime 和 i_ctime 分别为最后一次访问该文件的时间、修改该文件的时间以及最初创建该文件的时间。

对于具有数据部分的文件（磁盘文件或“普通文件”）来说，i_size 就是其数据部分当前的大小。至于数据所在的位置，则根据文件系统的不同而记录在 inode 中的 union 里面。

就像人可以有别名一样，文件也可以有多个文件名，也就是说可以将一个已经创建的文件“连接”（link）到另一个文件名。这个“别名”与原来的文件名可以在同一个目录中，也可以在不同的目录中，但是这些不同的目录项都指向同一个 inode。与此相应，在 inode 结构中有个计数器 i_link，用来记住这个文件有多少个这样的连接。同时，还有个队列头 i_dentry，用来构成一个 dentry 结构的队列，沿着这个队列就可以找到与这个文件相联系的所有 dentry 结构。

除了相对静态的信息以外，inode 结构中还有些成分用于表示一些动态的信息。例如，i_count 就是 inode 结构的共享计数，这个数值在系统运行的过程中是常常在变化的。又如，inode 结构可以通过它的几个 list_head 结构动态地链入到内存中的若干队列中，这种关系显然也是在动态地变化的。

另外，inode 结构中 union 里面的信息也有很多是动态的。显然，inode 结构中相对静态的一些信息是需要保存在“不挥发性”介质如磁盘上的。这一点对具有数据部分的磁盘文件固不待言，就是对于不具有数据部分的设备文件和特殊文件也是必需的（只有少数特殊文件例外，如无名管道文件）。所以，在前面的文件系统层次图（图 5.3）中，实际上从 VFS 和磁盘介质之间还应该加上一条连线，表示 VFS 层为管理的目的在磁盘上保存和恢复 inode 结构（以及其它一些数据结构，如 dentry）所需的一些信息。以后读者还会看到，磁盘的格式化也考虑到了这个问题。以 Ext2 格式为例，磁盘上的记录块（扇区）主要分成两部分，一部分用于索引节点，一部分用于文件的数据。给定一个索引节点号，就可以通过磁盘的设备驱动程序将其所在的记录块读入内存中。

那么，既然只要把 inode 结构中的部分信息保存在磁盘上的“索引节点”中，这些节点又是什么样的呢？这要看具体的文件系统而定。就 Linux 本身的文件系统 Ext2 而言，那就是 ext2_inode 数据结构，这是在 include/linux/ext2_fs.h 中定义的，读者不妨先大致上比较一下它与 inode 结构的异同。

==================== include/linux/ext2_fs.h 214 269 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
	__u16	i_mode;		/* File mode */
	__u16	i_uid;		/* Low 16 bits of Owner Uid */
	__u32	i_size;		/* Size in bytes */
	__u32	i_atime;	/* Access time */
	__u32	i_ctime;	/* Creation time */
	__u32	i_mtime;	/* Modification time */
	__u32	i_dtime;	/* Deletion Time */
	__u16	i_gid;		/* Low 16 bits of Group Id */
	__u16	i_links_count;	/* Links count */
	__u32	i_blocks;/* Blocks count */
	__u32	i_flags;	/* File flags */
	union {
		struct {
			__u32  l_i_reserved1;
		} linux1;
		struct {
			__u32  h_i_translator;
		} hurd1;
		struct {
			__u32  m_i_reserved1;
		} masix1;
	} osd1;				/* OS dependent 1 */
	__u32	i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
	__u32	i_generation;	/* File version (for NFS) */
	__u32	i_file_acl;	/* File ACL */
	__u32	i_dir_acl;	/* Directory ACL */
	__u32	i_faddr;	/* Fragment address */
	union {
		struct {
			__u8	l_i_frag;	/* Fragment number */
			__u8	l_i_fsize;	/* Fragment size */
			__u16	i_pad1;
			__u16	l_i_uid_high;	/* these 2 fields    */
			__u16	l_i_gid_high;	/* were reserved2[0] */
			__u32	l_i_reserved2;
		} linux2;
		struct {
			__u8	h_i_frag;	/* Fragment number */
			__u8	h_i_fsize;	/* Fragment size */
			__u16	h_i_mode_high;
			__u16	h_i_uid_high;
			__u16	h_i_gid_high;
			__u32	h_i_author;
		} hurd2;
		struct {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
263                 __u8 m_i_frag; /* Fragment number */
264                 __u8 m_i_fsize;     /* Fragment size */
265                 __u16     m_pad1;
266                 __u32     m_i_reserved2[2];
267             } masix2;
268         } osd2;                 /* OS dependent 2 */
269 };
```

同样，读者以后还会不时地需要回过来看看这个结构的定义。这里我们暂不解释这个结构中的成分，而只是指出：除 Linux 外，FSF 还打算在它的其它两个操作系统中也采用 Ext2 文件系统，但是在具体使用上又略有不同，所以在这个结构中有两个 union，即 osd1 和 osd2，都要视实际运行的操作系统而作不同的解释。

虽然在 inode 结构（以及 ext2_inode 结构）中包含了关于文件的组织和管理的信息，但是还有一项关键性的信息，即文件名，却并不在其内。显然，我们需要一种机制，使得根据一个文件的文件名就可以在磁盘上找到该文件的索引节点，从而在内存中建立起代表该文件的 inode 结构。这种机制就是文件系统的目录树。这棵倒立的“树”从系统的“根节点”，即“/”开始向下伸展，除最底层的“叶”节点为“文件”以外，其它的中间节点都是“目录”。其实，目录也是一种文件，是一种特殊的磁盘文件。这种文件的“文件名”就是目录名，也有索引节点，并且有数据部分。所不同的是，其数据部分的内容只包括“目录项”。对 Ext2 文件系统来说，这种“目录项”就是 ext2_dir_entry，后来改成了 ext2_dir_entry_2 数据结构（但保持兼容），它也是在 include/linux/ext2_fs.h 中定义的：

==================== include/linux/ext2_fs.h 474 474 ====================

```
474 #define EXT2_NAME_LEN 255
    . . . . . .
```

==================== include/linux/ext2_fs.h 483 495 ====================

```
483 /*
484  * The new version of the directory entry.  Since EXT2 structures are
485  * stored in intel byte order, and the name_len field could never be
486  * bigger than 255 chars, it's safe to reclaim the extra byte for the
487  * file_type field.
488  */
489 struct ext2_dir_entry_2 {
490     __u32     inode;              /* Inode number */
491     __u16     rec_len;        /* Directory entry length */
492     __u8 name_len;      /* Name length */
493     __u8 file_type;
494     char name[EXT2_NAME_LEN];    /* File name */
495 };
```

文件名（不包括路径部分）的最大长度为 255 个字符。老版本的 ext2_dir_entry 结构中 name_len 为无符号短整数，而在新版本的 ext2_dir_entry_2 中则改为 8 位的无符号字符，腾出一半用作文件类型 file_type。目前，已经定义的文件类型为：

==================== include/linux/ext2_fs.h 497 510 ====================

```
497 /*
498  * Ext2 directory file types.  Only the low 3 bits are used.  The
```

```
 * other bits are reserved for now.
 */
#define EXT2_FT_UNKNOWN     0
#define EXT2_FT_REG_FILE    1
#define EXT2_FT_DIR         2
#define EXT2_FT_CHRDEV      3
#define EXT2_FT_BLKDEV      4
#define EXT2_FT_FIFO        5
#define EXT2_FT_SOCK        6
#define EXT2_FT_SYMLINK     7

#define EXT2_FT_MAX         8
```

这里的 EXT2_FT_CHRDEV 和 EXT2_FT_BLKDEV 分别表示字符设备文件和块设备文件。我们在前面提到过的“/proc”下的特殊文件并不单独成为一类，而是作为常规文件，即 EXT2_FT_REG_FILE 出现在目录项中。至于最后怎样与真正的常规文件相区分，则读者在读完本章以后自会明白。

注意 ext2_dir_entry_2 结构中有个字段 rec_len，说明这个数据结构的长度并不是固定的。由于节点名的长度相差可以很大，固定按最大长度 255 分配空间会造成浪费，所以将这个数据结构的长度设计成可变的。当然，在这样的数据结构中其可变部分（这里是 name）必须放在最后。

磁盘上的 ext2_inode 数据结构在内存中的对应物为 inode 结构，但二者有很大不同；同样，目录项 ext2_dir_entry_2 在内存中的对应物是 dentry 结构，但是这二者也有很大不同。数据结构 dentry 是在 include/linux/dcache.h 中定义的：

==================== include/linux/dcache.h 57 77 ====================

```
#define DNAME_INLINE_LEN 16

struct dentry {
     atomic_t d_count;
     unsigned int d_flags;
     struct inode  * d_inode;    /* Where the name belongs to - NULL is negative */
     struct dentry * d_parent;   /* parent directory */
     struct list_head d_vfsmnt;
     struct list_head d_hash;    /* lookup hash list */
     struct list_head d_lru;     /* d_count = 0 LRU list */
     struct list_head d_child;   /* child of parent list */
     struct list_head d_subdirs; /* our children */
     struct list_head d_alias;   /* inode alias list */
     struct qstr d_name;
     unsigned long d_time;       /* used by d_revalidate */
     struct dentry_operations  *d_op;
     struct super_block * d_sb;  /* The root of the dentry tree */
     unsigned long d_reftime;    /* last time referenced */
     void * d_fsdata;        /* fs-specific data */
     unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
};
```

很明显，dentry 结构中的大部分成分都是动态信息。就是静态部分如文件名也与磁盘上的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

ext2_dir_entry_2 有很大的不同，相比之下几乎是面目全非。以后我们将结合代码解释其主要成分的用途。其实，dentry 与 ext2_dir_entry_2 之间以及 inode 与 ext2_inode 之间的这种显著不同并不奇怪，因为 dentry 和 inode 是属于 VFS 层的数据结构，需要适用于各种不同的文件系统；而 ext2_dir_entry_2 和 ext2_inode 则是专门针对 Ext2 文件系统而设计的，所以前者除包含了许多动态信息以外，还是对后者的一种抽象和扩充，并不只是后者的映象。

说到这里，读者可能会产生一个疑问：要访问一个文件就得先访问一个目录，才能根据文件名从目录中找到该文件的目录项，进而找到其 i 节点；可是目录本身也是文件，它本身的目录项又在另一个目录项中，这一来不是成了“先有鸡还是先有蛋”的问题，或者说递归了吗？这个圈子的出口在哪儿呢？我们不妨换一个方式来问这个问题，那就是：是否有这样一个目录，它本身的“目录项”不在其它目录中，而可以在一个固定的位置上或者通过一个固定的算法找到，并且从这个目录出发可以找到系统中的任何一个文件？答案是肯定的，这个目录就是系统的根目录“/”，或者说“根设备”上的根目录。每一个“文件系统”，即每一个格式化成某种文件系统的存储设备上都有一个根目录，同时又都有一个“超级块”（super block），根目录的位置以及文件系统的其它一些参数就记录在超级块中。超级块在设备上的逻辑位置都是固定的，例如，在磁盘上总是在第二个逻辑块（第一个逻辑块为引导块），所以不需要再从其它什么地方去“查找”。同时，对于一个特定的文件系统，超级块的格式也是固定的，系统在初始化时要将一个存储设备（通常就是从中引导出操作系统的那个设备）作为整个系统的“根设备”，它的根目录就成为整个文件系统的“总根”，就是“/”。更确切地说，就是把根设备的根目录“安装”在文件系统的总根“/”节点上。有了根设备以后，还可以进而把其它存储设备也安装到文件系统中空闲的目录节点上。所谓“安装”，就是从一个存储设备上读入超级块，在内存中建立起一个 super_block 结构。再进而将此设备上的根目录与文件系统中已经存在的一个空白目录挂上钩。系统初始化时整个文件系统只有一个空白目录“/”，所以根设备的根目录就安装在这个节点上。这样，从根目录“/”开始，根据给定的“全路径名”就可以找到文件系统中的任何一个文件，而不论这个文件是在哪一个存储设备上，只要文件所在的存储设备已经安装就行了。

但是，每次都要提供一个全路径名，并且每次都要从根目录“/”开始查找，既不方便也是一种浪费。所以系统也提供了从“当前目录”开始查找的手段。每一个进程在每一时刻都有一个“当前工作目录 pwd”，用户可以改变这个目录，但是永远都有这么个目录存在。这样，就可以只提供一个从 pwd 开始的“相对路径名”来查找一个文件。这就是前面看到过的 fs_struct 数据结构中为什么要有个指针 pwd 的原因。这个指针总是指向本进程的“当前工作目录”的 dentry 结构，而进程的 task_struct 结构中的指针 fs 则总是指向一个有效的 fs_struct 结构。每当一个进程通过 chdir()系统调用进入一个目录，或者在 login 进入用户的原始目录（“Home Directory”）时，内核就使该进程的 pwd 指针指向这个目录在内存中的 dentry 结构。相对路径名还可以用“../”开头，表示先向上找到当前目录的父目录，再从那里开始查找。相应地，在 dentry 结构中也有个指针 d_parent，指向其父目录的 dentry 结构。

如前所述，fs_struct 结构中还有一个指针 root，指向本进程的根目录“/”的 dentry 结构。前面讲过，“/”表示整个文件系统的总根，可这只是就一般而言，或者是对早期的 Unix 系统而言。事实上，特权用户可以通过一个系统调用 chroot()将另一个目录设置成本进程的根目录。从此以后，这个进程以及由这个进程所 fork()的子进程就把这个目录当成了文件系统的根，遇到文件的全路径名时就从这个目录而不是从真正的文件系统总根开始查找。例如，要是这个进程执行一个系统调用 chdir("/")，就会转到这个“现在”的根目录而不是真正的根目录。这种特殊的设计也是从实践需求引起的，最初是为了

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

克服 FTP，特别是匿名 FTP 的一个安全性问题。FTP 的服务进程（所谓“守护神 ” daemon）是特权用户进程。当一个远程的用户与 FTP 服务进程建立起连接以后，就可以在远地发出诸如 “ cd /”、“ get /etc/passwd”之类的命令。显然，这给系统的安全性造成了一个潜在的缺口，现在有了进程自己的“根目录”以及系统调用 chroot()，就可以让 FTP 服务进程把另一目录当成它的根目录，从而当远程用户要求 “ get /etc/passwd”时就会得到“文件找不到”之类的出错信息，从而保证了 passwd 口令文件的安全性。

而且，fs_struct 结构中还有一个指针 altroot，指向本进程的“替换根目录”。当进程执行一个系统调用 chdir("/")时，如果它有替换根目录，即指针 altroot 不为 0，就会转入其替换根目录，否则才转入其实在根目录。这样就可以视具体的情况而在两个“根目录”中切换，让用户在不同的情况下“看到 ”不同的根目录。

对于普通文件，文件系统层最终要通过磁盘或其它存储设备的驱动程序从存储介质上读或写。就 Ext2 文件系统而言，从磁盘文件的角度来看，对存储介质的访问可以涉及到四种不同的目标，那就是：

(1) 文件中的数据，包括目录的内容，即目录项 ext2_dir_entry_2 数据结构。
(2) 文件的组织与管理信息，即索引节点 ext2_inode 数据结构。
(3) 磁盘的超级块。如果物理的磁盘被划分成若干分区，那就包括每个“逻辑磁盘”的超级块。
(4) 引导块。

每个按 Ext2 格式经过格式化的磁盘(或逻辑盘)存储介质都相应地被划分成至少 4 个部分(图 5.4)。其中引导块永远是介质上的第一个记录块，超级块永远是介质上的第二个记录块，其它两部分的大小则取决于磁盘大小等参数，这些参数都存储在超级块中。

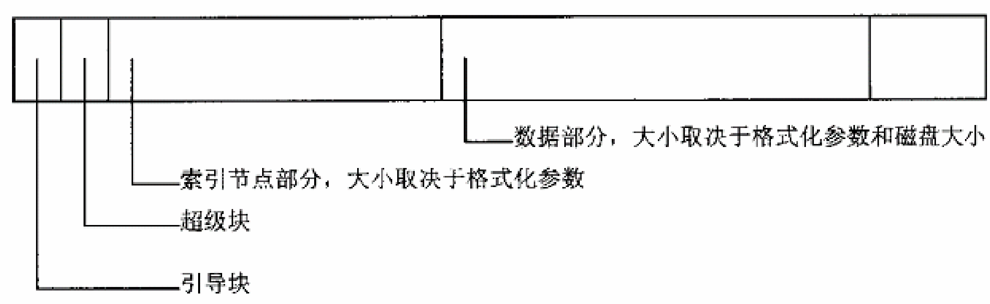


图 5.4 磁盘（逻辑盘）划分示意图

有的文件系统并没有索引节点这么一种数据结构，甚至没有这么一种概念。但是既然构成一个文件系统就必然存在着某种索引机制，从这种机制中就可以抽象出（或变换成）super_block 结构和 inode 结构中的公共信息。同时，super_block 结构也和 inode 结构一样包含着一个 union，对这一部分信息要根据具体的文件系统而加以不同的解释和使用。

从磁盘驱动程序的角度来看，则整个介质只是一个由若干记录块组成的一维阵列（记录块数组）而已，所以这种设备称为“块设备”。当文件系统层要从磁盘上读出一个索引节点时，要根据索引节点号和超级块中提供的信息，计算出这个索引节点在磁盘上的哪一个记录块以及在此记录块中的相对位移。然后，通过磁盘驱动程序读入这个记录块后再根据索引节点在记录块中的相对位移找到这个节点。如前所述，磁盘上的“根目录”是特殊的，其索引节点号保存在该磁盘的超级块中。从磁盘读一个特

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

定文件的内容（数据）则要稍为麻烦一点。先要读入该文件的索引节点，然后根据索引节点中提供的信息将数据在文件中的位移换算成磁盘上的记录块号，再通过磁盘驱动程序从磁盘上读入。

相比之下，作为“设备文件”的磁盘则不存在（或看不见）这样的逻辑划分，而只是将磁盘看成一个巨大的线性存储空间（字节数组）。当从作为设备文件的磁盘读出时，只要将数据在此文件中的位移换算成磁盘上的记录块号，就可以通过磁盘驱动程序读入了。不过，在此之前也要先找到代表着这个设备文件的目录项和索引节点，才能把字符串形式的设备文件名转换成驱动程序所需要的设备号。

在前面我们曾把具体的文件系统比喻作“接口卡”，而把虚拟文件系统 VFS 比喻成一条插槽。因此，file 结构中的指针 f_op 就可以看作插槽中的一个触点，并且在 dentry、inode 等结构中都有着类似的触点。所以，如果把整个具体文件系统比喻成“接口卡”的话，那么这种接口卡的“插槽”分成好几段，而 file 结构只是其中最主要的一段。有关的数据结构有：

(1) 文件操作跳转表，即 file_operations 数据结构：file 结构中的指针 f_op 指向具体的 file_operations 结构，这是 read()、wirte()等文件操作的跳转表。一种文件系统并不只限于一个 file_operations 结构，如 Ext2 就有两个这样的数据结构，分别用于普通文件和目录文件。
(2) 目录项操作跳转表，即 dentry_operations 数据结构：dentry 结构中的指针 d_op 指向具体的 dentry_operations 数据结构，这是内核中 hash()、compare()等内部操作的跳转表。如果 d_op 为 0 则表示按 Linux 默认的（即 Ext2）方式办。注意，这里说的是目录项，而不是目录，目录本身是一种特殊用途和具有特殊结构的文件。
(3) 索引节点操作跳转表，即 inode_operations 数据结构：inode 结构中的指针 i_op 指向具体的 inode_operations 数据结构，这是 mkdir()、mknod()等文件操作以及 lockup()、permission()等内部函数的跳转表。同样，一种文件系统也并不只限于一个 file_operations 结构。
(4) 超级块操作跳转表，即 super_operations 数据结构：super_block 结构中的指针 s_op 指向具体的 super_operations 数据结构，这是 read_inode()、write_inode()、delete_inode()等内部操作的跳转表。
(5) 超级块本身也因文件系统而异。

由此可见，file 结构、dentry 结构、inode 结构、super_block 结构以及关于超级块位置的约定都属于 VFS 层。

此外，inode 结构中还有一个指针 i_fop，也指向具体的 file_operations 数据结构，实际上 file 结构中的指针 f_op 只是 inode 结构中这个指针的一个副本，在打开文件的时候从目标文件的 inode 结构中复制到 file 结构中。

最后还要指出，虽然每个文件都有目录项和索引节点在磁盘上，但是只有在需要时才在内存中为之建立起相应的 dentry 和 inode 数据结构。

5.2 从路径名到目标节点

本节先介绍几个函数的代码，主要是两个函数，即 path_init()和 path_walk()以及它们下面的一些低层函数。目的在于帮助读者加深对文件系统内部结构的理解，同时也为以后的代码阅读做些准备，因

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

为以这两个函数为入口的操作比较大，并且很重要，在本章后面几节中常常要用到。这两个函数通常都是连在一起调用的，二者合在一起就可以根据给定的文件路径名在内存中找到或建立代表着目标文件或目录的 dentry 结构和 inode 结构。在老一些的版本中，这一部分功能一直是通过一个叫 namei()（后来加了一个叫 lnamei()）的函数完成的，现在则有了新的实现。与 namei()和 lnamei()相对应，现在有一个函数__user_walk()将 path_init()和 path_walk()“包装”在一起。不过，内核代码中直接调用这两个函数的地方也有不少。本节涉及的代码基本上都在文件 fs/namei.c 中。

先看“外包装”，即__user_walk()：

==================== fs/namei.c 778 803 ====================

```
/*
 *   namei()
 *
 * is used by most simple commands to get the inode of a specified name.
 * Open, link etc use their own routines, but this is enough for things
 * like 'chmod' etc.
 *
 * namei exists in two versions: namei/lnamei. The only difference is
 * that namei follows links, while lnamei does not.
 * SMP-safe
 */
int __user_walk(const char *name, unsigned flags, struct nameidata *nd)
{
     char *tmp;
     int err;

     tmp = getname(name);
     err = PTR_ERR(tmp);
     if (!IS_ERR(tmp)) {
          err = 0;
          if (path_init(tmp, flags, nd))
               err = path_walk(tmp, nd);
          putname(tmp);
     }
     return err;
}
```

其中调用参数 name 指向在用户空间中的路径名；flags 的内容则是一些标志位，定义于文件 include/linux/fs.h：

==================== include/linux/fs.h 1128 1140 ====================

```
/*
 * The bitmask for a lookup event:
 *  - follow links at the end
 *  - require a directory
 *  - ending slashes ok even for nonexistent files
 *  - internal "there are more path compnents" flag
 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define LOOKUP_FOLLOW       (1)
#define LOOKUP_DIRECTORY    (2)
#define LOOKUP_CONTINUE     (4)
#define LOOKUP_POSITIVE     (8)
#define LOOKUP_PARENT       (16)
#define LOOKUP_NOALT        (32)
```

这些标志位都是对怎样寻找目标的指示。例如，LOOKUP_DIRECTORY 表示要寻找的目标必须是个目录；而 LOOKUP_FOLLOW 表示如果找到的目标只是“符号连接”到其它文件或目录的一个目录项，则要顺着连接链一直找到终点。所谓“连接”是指一个“节点”（目录项或文件）直接指向另一个节点，成为另一个节点的代表。注意，“符号连接”与普通连接不同，普通的连接只能建立在同一个存储设备上，而“符号连接”可以是跨设备的；内核提供了两个不同的系统调用 link()和 symlink()，分别用于普通连接和“符号连接”的建立。由于“符号连接”可以是跨设备的，所以其终点有可能“悬空”，而普通连接的终点则必定是落实的。当路径中包含着“符号连接”时，对于是否继续顺着连接链往下搜索，则另有一些附加规定，对此，代码的作者在注释中加了说明（fs/namei.c）：

==================== fs/namei.c 87 94 ====================

```
/* [Feb-Apr 2000 AV] Complete rewrite. Rules for symlinks:
 *   inside the path - always follow.
 *   in the last component in creation/removal/renaming - never follow.
 *   if LOOKUP_FOLLOW passed - follow.
 *   if the pathname has trailing slashes - follow.
 *   otherwise - don't follow.
 * (applied in that order).
 *
```

==================== fs/namei.c 100 100 ====================

```
 */
```

注释中谈到，如果在一个路径名内部的某个中间节点是符号连接，那就总是要跟随（follow）；而在创建 / 删除 / 改名操作中如果路径名的最后一个节点是符号连接则不要跟随（读者不妨想想为什么？）。

至于其它一些标志位的用途，在阅读代码的过程中自会碰到。此处要提醒读者注意：并非所有标志位对所有文件系统都有意义。

最后一个参数 nd 是个结构指针，数据结构 nameidata 的定义也在 include/linux/fs.h 中：

==================== include/linux/fs.h 613 619 ====================

```
struct nameidata {
     struct dentry *dentry;
     struct vfsmount *mnt;
     struct qstr last;
     unsigned int flags;
     int last_type;
};
```

这种数据结构是临时性的，只用来返回搜索的结果。成功返回时，其中的指针 dentry 指向所找到的 dentry 结构，而在该 dentry 结构中则有指针指向相应的 inode 结构。指针 mnt 则指向一个 vfsmount

数据结构，它记录着所属文件系统的安装信息，例如文件系统的安装点、文件系统的根节点等等。

回到__user_walk()，先通过 getname()在系统空间中分配一个页面，并从用户空间把文件名复制到这个页面中。由于分配的是一个页面，所以整个路径名可以长达 4K 字节。同时，因为这块空间是动态分配的，所以在使用完以后要通过 putname()将其释放。代码中用到的 PTR_ERR 和 IS_ERR 都是 inline 函数，均在 include/linux/fs.h 中：

==================== include/linux/fs.h 1105 1126 ====================

```
/*
 * Kernel pointers have redundant information, so we can use a
 * scheme where we can return either an error code or a dentry
 * pointer with the same return value.
 *
 * This should be a per-architecture thing, to allow different
 * error and pointer decisions.
 */
static inline void *ERR_PTR(long error)
{
     return (void *) error;
}

static inline long PTR_ERR(const void *ptr)
{
     return (long) ptr;
}

static inline long IS_ERR(const void *ptr)
{
     return (unsigned long)ptr > (unsigned long)-1000L;
}
```

这样，剩下的就是紧挨在一起的 path_init()和 path_walk()两个函数了。先在 fs/namei.c 文件中看 path_init()的代码：

==================== fs/namei.c 690 702 ====================

```
/* SMP-safe */
int path_init(const char *name, unsigned int flags, struct nameidata *nd)
{
     nd->last_type = LAST_ROOT; /* if there are only slashes... */
     nd->flags = flags;
     if (*name=='/')
          return walk_init_root(name,nd);
     read_lock(&current->fs->lock);
     nd->mnt = mntget(current->fs->pwdmnt);
     nd->dentry = dget(current->fs->pwd);
     read_unlock(&current->fs->lock);
     return 1;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

首先将 nameidata 结构中的 last_type 字段设置成 LAST_ROOT。这个字段可能有的值定义于 include/linux/fs.h 中：

==================== include/linux/fs.h 1141 1144 ====================

```
/*
 * Type of the last component on LOOKUP_PARENT
 */
enum {LAST_NORM, LAST_ROOT, LAST_DOT, LAST_DOTDOT, LAST_BIND};
```

在搜索的过程中，这个字段的值会随路径名的当前搜索结果而变。例如：如果成功地找到了目标文件，那么这个字段的值就变成了 LAST_NORM；而如果最后停留在一个“.”上，则变成 LAST_DOT。

下面就取决于路径名是否以“/”开头了。

我们先看相对路径名，即不以“/”开头时的情况。以前讲过，进程的 task_struct 结构中有个指针 fs 指向一个 fs_struct 结构。在 fs_struct 结构中有个指针 pwd 指向进程的“当前工作目录”的 dentry 结构。相对路径是从当前工作目录开始的，所以将 nameidata 结构中的指针 dentry 也设置成指向这个当前工作目录的 dentry 结构，表示在虚拟的绝对路径中这个节点以及所有在此之前的节点都已经解决了。同时，这个具体的 dentry 结构现在多了一个“用户”，所以要调用 dget()递增其共享计数。除此以外，fs_sturct 结构中还有个指针 pwdmnt 指向一个 vfsmount 结构。每当将一个存储设备（或称“文件系统”）安装到现有文件系统中的某个节点（空白目录，）时，内核就要为之建立起一个 vfsmount 结构，这个结构中既包含着有关该设备（或者说“子系统”）的信息，也包含了有关安装点的信息。系统中的每个文件系统，包括根设备上的文件系统，都要经过安装，所以 fs_sturct 结构中的指针 pwdmnt 总是指向一个 vfsmount 结构。详情可参阅后面“文件系统的安装与拆卸”一节。相应地，在 nameidata 结构中也有个指针 mnt，要把它设置成指向同一个 vfsmount 结构。这样，对路径搜索的准备工作，即对 nameidata 结构的初始化就完成了。

可是，如果路径名是以“/”开头的绝对路径，那就要通过 walk_init_root()从根节点开始查找（fs/namei.c）：

==================== fs/namei.c 671 688 ====================

[path_init()>walk_init_root()]

```
/* SMP-safe */
static inline int
walk_init_root(const char *name, struct nameidata *nd)
{
	read_lock(&current->fs->lock);
	if (current->fs->altroot && !(nd->flags & LOOKUP_NOALT)) {
		nd->mnt = mntget(current->fs->altrootmnt);
		nd->dentry = dget(current->fs->altroot);
		read_unlock(&current->fs->lock);
		if (__emul_lookup_dentry(name,nd))
			return 0;
		read_lock(&current->fs->lock);
	}
	nd->mnt = mntget(current->fs->rootmnt);
	nd->dentry = dget(current->fs->root);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    read_unlock(&current->fs->lock);
    return 1;
}
```

如果当前进程并未通过 chroot()系统调用设置自己的“替换”根目录，则代码中 if 语句里的 current->fs->altroot 为 0，所以把 nameidata 中的两个指针分别设置成指向当前进程的根目录的 dentry 结构及其所在设备的 vfsmount 结构。反之，如果已经设置了“替换”根目录，那就要看当初调用 path_init()时参数 flags 中的标志位 LOOKUP_NOALT 是否为 1 了。通常这个标志位为 0，所以如果已经设置了“替换”根目录就会通过__emul_lookup_dentry()将 nameidata 结构中的指针设置成指向“替换”根目录。

这“替换”根目录到底是怎么回事呢？原来，在有些 Unix 变种（如 Solaris 等）中，可以在文件系统中（通常是在“ /usr”下面）创建一棵子树，例如“ /usr/altroot/home/userl/… ”。然后，当用户调用 chroot()设置其自己的根目录时，系统会自动将该进程的 fs_struct 结构中的 altroot 和 altrootmnt 两个指针设置成给定路径名在前述子树中的对应节点，那个对应节点就成了“替换”根目录。不过在 i386 处理器上的 linux 目前并不支持这种功能，所以这里 if 语句中的 current->fs->altroot 总是 NULL，因而不起作用。

从 path_init()成功返回时，nameidata 结构中的指针 dentry 指向路径搜索的起点，接着就是通过 path_walk()顺着路径名的指引进行搜索了。这个函数比较大，所以我们逐段地往下看（fs/namei.c）：

==================== fs/namei.c 414 437 ====================

```
/*
 * Name resolution.
 *
 * This is the basic name resolution function, turning a pathname
 * into the final dentry.
 *
 * We expect 'base' to be positive and a directory.
 */
int path_walk(const char * name, struct nameidata *nd)
{
    struct dentry *dentry;
    struct inode *inode;
    int err;
    unsigned int lookup_flags = nd->flags;

    while (*name=='/')
        name++;
    if (!*name)
        goto return_base;

    inode = nd->dentry->d_inode;
    if (current->link_count)
        lookup_flags = LOOKUP_FOLLOW;

```

如果路径名是以“ /”开头的，就把它跳过去，因为在这种情况下 nameidata 结构中的指针 dentry 已经指向本进程的根目录了。注意，多个连续的“ /”与一个“ /”字符是等价的。如果路径名中仅仅含有“ /”字符的话，那么其目标就是根目录，所以任务已经完成，可以返回了。不然，就继续搜索。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进程的 task_struct 结构中有个计数器 link_count。在搜索过程中有可能碰到一个节点（目录项）只是指向另一个节点的连接，此时就用这个计数器来对链的长度进行计数，这样，当链的长度达到某一个值时就可以终止搜索而失败返回，以防陷入循环。另一方面，当顺着“符号连接”进入另一个设备上的文件系统时，有可能会递归地调用 path_walk()。所以，进入 path_walk()后，如果发现这个计数值非 0，那就表示正在顺着“符号连接”递归调用 path_walk()往前搜索的过程中，此时不管怎样都把 LOOKUP_FOLLOW 标志位设成 1。这里还要指出，作为 path_walk()起点的节点必定是一个目录，一定有相应的索引节点存在，所以指针 inode 一定是有效的，而不可能是空指针。

接下去是一个对路径中的节点所作的 for 循环，由于循环体较大，我们也只好分段来看(fs/namei.c)。

==================== fs/namei.c 438 467 ====================

[path_walk()]

```
	/* At this point we know we have a real path component. */
	for(;;) {
		unsigned long hash;
		struct qstr this;
		unsigned int c;

		err = permission(inode, MAY_EXEC);
		dentry = ERR_PTR(err);
		if (err)
			break;

		this.name = name;
		c = *(const unsigned char *)name;

		hash = init_name_hash();
		do {
			name++;
			hash = partial_name_hash(c, hash);
			c = *(const unsigned char *)name;
		} while (c && (c != '/'));
		this.len = name - (const char *) this.name;
		this.hash = end_name_hash(hash);

		/* remove trailing slashes? */
		if (!c)
			goto last_component;
		while (*++name == '/');
		if (!*name)
			goto last_with_slashes;

```

首先检查当前进程对当前节点的访问权限。函数 permission()的代码与作用请参阅“访问权限与文件安全性”一节。这里所检查的是对路径中各层目录（而不是目标文件）的访问权限。注意，对于中间节点所需的权限为“执行”权，即 MAY_EXEC。如果权限不符，则 permission 返回一个出错代码，从而通过 break 语句结束循环，搜索就失败了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

循环体中的局部量 this 是个 qstr 数据结构，用来存放路径名中当前节点的杂凑值以及节点名的长度，这个数据结构的定义在 include/linux/dcache.h 中：

==================== include/linux/dcache.h 20 28 ====================

```
/*
 * "quick string" -- eases parameter passing, but more importantly
 * saves "metadata" about the string (ie length and the hash).
 */
struct qstr {
	const unsigned char * name;
	unsigned int len;
	unsigned int hash;
};
```

回到代码中的第 453～457 行，这几行的作用就是逐个字符地计算出当前节点名的杂凑值，至于具体的杂凑函数，我们就不关心了。

路径名中的节点是以“/”字符分隔的，所以紧随当前节点名的字符只有两种可能：

(1) 是“\0”，就是说当前节点已经是路径名中的最后一节，所以转入 last_component。

(2) 是个“/”字符，这里又有两种可能，第一种情况是当前节点实际上已是路径名中的最后一个节点，只不过在此后面又多添了若干个“/”字符。这种情况常常发生在用户界面上，特别是在 shell 的命令行中，例如“ls /usr/include/”，这是允许的。但是当然最后的节点必须是个目录，所以此时转到 last_with_slashes。第二种情况就是当前节点为中间节点（包括起始节点），所以“/”字符（或者接连若干个“/”字符）后面还有其它字符。这种情况下就将其跳过，继续往下执行。

现在，要回过头来看当前节点了。记住，这个节点一定是中间节点或起始节点（否则就转到 last_component 去了），这种节点一定是个目录。对于代表着文件的节点名来说，以“.”开头表示这是个隐藏的文件，而对于代表着目录的节点名则只有在两种情况下才是允许的。一种是节点名为“.”，表示当前目录，即不改变目录。另一种就是“..”，表示当前目录的父目录。

继续往下看：

==================== fs/namei.c 468 484 ====================

[path_walk()]

```
		/*
		 * "." and ".." are special - ".." especially so because it has
		 * to be able to know about the current root directory and
		 * parent relationships.
		 */
		if (this.name[0] == '.') switch (this.len) {
			default:
				break;
			case 2:
				if (this.name[1] != '.')
					break;
				follow_dotdot(nd);
				inode = nd->dentry->d_inode;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
481                     /* fallthrough */
482                 case 1:
483                     continue;
484             }
```

就是说，如果当前节点名的第一个字符是“.”，则节点名的长度只能是 1 或者 2，并且当长度为 2 时第二个字符也必须是“.”；否则搜索就失败了（见 475 行和 478 行的 break 语句）。

如果当前节点名真的是“..”，那就要往上跑到当前已经到达的节点 nd->dentry 的父目录去。这是由 follow_dotdot()完成的：

==================== fs/namei.c 380 413 ====================

[path_walk()>follow_dotdot()]

```
static inline void follow_dotdot(struct nameidata *nd)
{
    while(1) {
        struct vfsmount *parent;
        struct dentry *dentry;
        read_lock(&current->fs->lock);
        if (nd->dentry == current->fs->root &&
            nd->mnt == current->fs->rootmnt)  {
            read_unlock(&current->fs->lock);
            break;
        }
        read_unlock(&current->fs->lock);
        spin_lock(&dcache_lock);
        if (nd->dentry != nd->mnt->mnt_root) {
            dentry = dget(nd->dentry->d_parent);
            spin_unlock(&dcache_lock);
            dput(nd->dentry);
            nd->dentry = dentry;
            break;
        }
        parent=nd->mnt->mnt_parent;
        if (parent == nd->mnt) {
            spin_unlock(&dcache_lock);
            break;
        }
        mntget(parent);
        dentry=dget(nd->mnt->mnt_mountpoint);
        spin_unlock(&dcache_lock);
        dput(nd->dentry);
        nd->dentry = dentry;
        mntput(nd->mnt);
        nd->mnt = parent;
    }
}
```

但是这里又要分三种情况：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第一种情况，已到达节点 nd->dentry 就是本进程的根节点，这时不能再往上跑了，所以保持 nd->dentry 不变。

第二种情况，已到达节点 nd->dentry 与其父节点在同一个设备上。在这种情况下，既然已经到达的这个节点的 dentry 结构已经建立，则其父节点的 dentry 结构也必然已经建立在内存中，而且 dentry 结构中的指针 d_parent 就指向其父节点，所以往上跑一层是很简单的事情。

最后一种情况，已到达节点 nd->dentry 就是其所在设备上的根节点，往上跑一层就要跑到另一个设备上去了。如前所述，当将一个存储设备“安装”到另一个设备上的某个节点时，内核会分配和设置一个 vfsmount 结构，通过这个结构将两个设备以及两个节点联结起来（详见“文件系统的安装与拆卸”）。所以，每个已经安装的存储设备（包括根设备）都有一个 vfsmount 结构，结构中有个指针 mnt 指向其“父设备”，但是根设备的这个指针则指向其自己，因为它再没有“父设备”了，而另一个指针 mnt_mountpoint，则指向代表着安装点（一定是个目录）的 dentry 结构。从文件系统的角度来看，安装点与所安装设备的根目录是等价的。我们已经在当前设备的根目录中，所以从这里往上跑一层就是要跑到安装点的上一层目录中（而不是安装点本身）。

先检查当前的 vfsmount 结构是否代表着根设备，如果是的话，立即就通过 399 行的 break 语句结束 while(1)循环。这样，nameidata 结构中的 dentry 和 mnt 两个指针就维持不变。这种情况相当于在根目录中打入命令“ cd ..”，或者“ cd usr/./../..”等等，读者不妨实验一下，看看结果如何。

反之，要是当前设备不是根设备，那就把 nameidata 结构中的两个指针分别设置成指向上层设备的 vfsmount 结构以及该设备上的安装点的上一层目录（dentry 结构），然后回到 while(1)循环的开始处。一般来说，安装点不会是一个设备上的根目录，所以这一次循环会将 nameidata 结构中的指针 dentry 指向安装点的父目录。可是，万一安装点真的就是上一层设备上的根目录（当然，必定是空的）呢？那也不要紧，只不过是再循环一次，再往上跑一层而已。

回到 path_walk()的代码中，注意“ case 2”的末尾没有 break 语句，所以会落入“ case 1”中通过 continue 语句回到 for(;;)循环的开头，继续处理路径中的下一个节点名。

当然，多数情况下节点名都不是以“ .”开头的，就是说多数情况下总是顺着路径名逐层往下跑，而不是往上跑的。我们继续往下看对“正常”节点名的处理流程：

==================== fs/namei.c 485 535 ====================

[path_walk()]

```
		/*
		 * See if the low-level filesystem might want
		 * to use its own hash..
		 */
		if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
			err = nd->dentry->d_op->d_hash(nd->dentry, &this);
			if (err < 0)
				break;
		}
		/* This does the actual lookups.. */
		dentry = cached_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
		if (!dentry) {
			dentry = real_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
			err = PTR_ERR(dentry);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
			if (IS_ERR(dentry))
				break;
		}
		/* Check mountpoints.. */
		while (d_mountpoint(dentry) && __follow_down(&nd->mnt, &dentry))
			;

		err = -ENOENT;
		inode = dentry->d_inode;
		if (!inode)
			goto out_dput;
		err = -ENOTDIR;
		if (!inode->i_op)
			goto out_dput;

		if (inode->i_op->follow_link) {
			err = do_follow_link(dentry, nd);
			dput(dentry);
			if (err)
				goto return_err;
			err = -ENOENT;
			inode = nd->dentry->d_inode;
			if (!inode)
				break;
			err = -ENOTDIR;
			if (!inode->i_op)
				break;
		} else {
			dput(nd->dentry);
			nd->dentry = dentry;
		}
		err = -ENOTDIR;
		if (!inode->i_op->lookup)
			break;
		continue;
		/* here ends the main loop */

```

有些文件系统通过 dentry_operations 结构中的指针 d_hash 提供它自己专用的杂凑函数，所以在这种情况下（可能已经转到另一个文件系统中了）就通过这个函数再计算一遍当前节点的杂凑值。

至此，所有的准备工作都已完成，接下去就要开始搜索了。

对当前节点的搜索是通过 cached_lookup()和 real_lookup()两个函数进行的。先通过 cache_lookup()在内存中寻找该节点业已建立的 dentry 结构。内核中有个杂凑表 dentry_hashtable，是一个 list_head 指针数组，一旦在内存中建立起一个目录节点的 dentry 结构，就根据其节点名的杂凑值挂入杂凑表中的某个队列，需要寻找时则还是根据杂凑值从杂凑表着手。当路径名中的某个节点变成 path_walk()的当前节点时，位于其“上游”的所有节点必定都已经有 dentry 结构在内存中，而当前节点本身及其“下游”的节点则不一定。如果在内存中找不到当前节点的 dentry 结构，那就要进一步通过 real_lookup()

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

到磁盘上通过其所在的目录寻找，找到后在内存中为其建立起 dentry 结构并将之挂入杂凑表中的某个队列。

内核中还有一个队列 dentry_unused，凡是已经没有用户，即其享计数为 0 的 dentry 结构就通过结构中的另一个 list_head 挂入这个队列。这个队列是一个 LRU 队列，当需要回收已经不在使用中的 dentry 结构的空间时，就从这个队列中找到已经空闲最久的 dentry 结构，再把这个结构从杂凑表队列中脱链而加以释放。所以，dentry_unused 是为缓冲存储而设置的辅助性的队列。不过，在一些特殊的情况下，可能会把一个还在使用中的 dentry 结构从杂凑表中脱链，迫使以后要访问这个节点的进程重新根据磁盘上的内容另行构筑一个 dentry 结构，而已经脱链的那个数据结构则由最后调用 dput()使其共享计数变成 0 的进程负责将其释放。

事实上，dentry 结构中有 6 个 list_head，即 d_vfsmnt、d_hash、d_lru、d_child、d_subdirs 和 d_alias。注意 list_head 既可以用来作为一个队列的头部，也可以用来将其所在的数据结构挂入到某个队列中。其中 d_vfsmnt 仅在该 dentry 结构为一安装点时才使用。一个 dentry 结构一经建立就通过其 d_hash 挂入杂凑表 dentry_hashtable 中的某个队列里，当共享计数变成 0 时则通过 d_lru 挂入 LRU 队列 dentry_unused 中。同时，dentry 结构通过 d_child 挂入在其父节点（上一层目录）的 d_subdirs 队列中，同时又通过指针 d_parent 指向其父目录的 dentry 结构。而它自己各个子目录的 dentry 结构则在它本身的 d_subdirs 队列中。

一个有效的 dentry 结构必定有一个相应的 inode 结构，这是因为一个目录项要么就代表着一个文件，要么就代表着一个目录，而目录实际上也是文件。所以，只要是有效的 dentry 结构，则其指针 d_inode 必定指向一个 inode 结构。可是，反过来一个 inode 却可能对应着不止一个 dentry 结构，也就是说，一个文件可以有不止一个文件名（或路径名）。这是因为一个已经建立的文件可以被连接（link）到其它文件名。所以，在 inode 结构中有个队列 i_dentry，凡是代表着这个文件的所有目录项都通过其 dentry 结构中的 d_alias 挂入相应 inode 结构中的 i_dentry 队列。此外，dentry 结构中还有指针 d_sb，指向其所在设备的超级块的 super_block 数据结构，以及指针 d_op，指向特定文件系统（指文件格式）的 dentry_operations 结构。也许可以说，dentry 结构是文件系统的核心数据结构，也是文件访问和为文件访问而做的文件路径搜索操作的枢纽。

下面是一个简要的总结：

- 每个 dentry 结构都通过队列头 d_hash 链入杂凑表 dentry_hashtable 中的某个队列里。
- 共享计数为 0 的 dentry 结构都通过队列头 d_lru 链入 LRU 队列 dentry_unused，在队列中等待释放或者“东山再起”。
- 每个 dentry 结构都通过指针 d_inode 指向一个 inode 数据结构。但是多个 dentry 结构可以指向同一个 inode 数据结构。
- 指向同一个 inode 数据结构的 dentry 结构都通过队列头 d_alias 链接在一起，都在该 inode 结构的 i_dentry 队列中。
- 每个 dentry 结构都通过指针 d_parent 指向其父目录节点的 dentry 结构，并通过队列头 d_child 跟同一目录中的其它节点的 dentry 结构链接在一起，都在父目录节点的 d_subdirs 队列中。
- 每个 dentry 结构都通过指针 d_sb 指向一个 super_block 数据结构。
- 每个 dentry 结构都通过指针 d_op 指向一个 dentry_operations 数据结构。
- 每个 dentry 结构都有个队列头 d_vfsmnt，用于文件系统的安装，详见“文件系统的安装和拆

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

卸”。

接下去我们看 cached_lookup()的代码（fs/namei.c）：

==================== fs/namei.c 243 258 ====================
[path_walk()>cached_lookup()]

```
/*
 * Internal lookup() using the new generic dcache.
 * SMP-safe
 */
static struct dentry * cached_lookup(struct dentry * parent, struct qstr * name, int flags)
{
     struct dentry * dentry = d_lookup(parent, name);

     if (dentry && dentry->d_op && dentry->d_op->d_revalidate) {
          if (!dentry->d_op->d_revalidate(dentry, flags) && !d_invalidate(dentry)) {
               dput(dentry);
               dentry = NULL;
          }
     }
     return dentry;
}
```

这里主要是通过 d_lookup()，在杂凑表中寻找，其代码在 fs/dcache.c 中：

==================== fs/dcache.c 703 749 ====================
[path_walk()>cached_lookup()>d_lookup()]

```
/**
 * d_lookup - search for a dentry
 * @parent: parent dentry
 * @name: qstr of name we wish to find
 *
 * Searches the children of the parent dentry for the name in question. If
 * the dentry is found its reference count is incremented and the dentry
 * is returned. The caller must use d_put to free the entry when it has
 * finished using it. %NULL is returned on failure.
 */

struct dentry * d_lookup(struct dentry * parent, struct qstr * name)
{
     unsigned int len = name->len;
     unsigned int hash = name->hash;
     const unsigned char *str = name->name;
     struct list_head *head = d_hash(parent,hash);
     struct list_head *tmp;

     spin_lock(&dcache_lock);
     tmp = head->next;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
724      for (;;) {
725          struct dentry * dentry = list_entry(tmp, struct dentry, d_hash);
726          if (tmp == head)
727              break;
728          tmp = tmp->next;
729          if (dentry->d_name.hash != hash)
730              continue;
731          if (dentry->d_parent != parent)
732              continue;
733          if (parent->d_op && parent->d_op->d_compare) {
734              if (parent->d_op->d_compare(parent, &dentry->d_name, name))
735                  continue;
736          } else {
737              if (dentry->d_name.len != len)
738                  continue;
739              if (memcmp(dentry->d_name.name, str, len))
740                  continue;
741          }
742          __dget_locked(dentry);
743          dentry->d_flags |= DCACHE_REFERENCED;
744          spin_unlock(&dcache_lock);
745          return dentry;
746      }
747      spin_unlock(&dcache_lock);
748      return NULL;
749  }
```

参数 parent 指向上一层节点的 dentry 结构，而 name 指向刚才在 path_walk 中建立的 qstr 结构。首先是要根据节点名的杂凑值从杂凑表中找到相应的队列指针，本来，以已经计算好的杂凑值作为下标从 list_head 指针数组 dentry_hashtable 中找到相应的表项是再简单不过的，可是这里（719 行）还要通过一个函数 d_hash()来做这件事，让我们来看看为什么：

==================== fs/dcache.c 696 701 ====================

[path_walk()>cached_1ookup()>d_lookup()>d_hash()]

```
696  static inline struct list_head * d_hash(struct dentry * parent, unsigned long hash)
697  {
698      hash += (unsigned long) parent / L1_CACHE_BYTES;
699      hash = hash ^ (hash >> D_HASHBITS) ^ (hash >> D_HASHBITS*2);
700      return dentry_hashtable + (hash & D_HASHMASK);
701  }
```

就是说，在已经根据节点名计算好的杂凑值基础上还要再进行一次杂凑，把父节点的 dentry 结构的地址也结合进杂凑值中。这无疑是很巧妙的做法。试想一下学校的计算机实验室，那里的系统可能为上百个学生分别在/home 下面建立了子目录，而每个学生的子目录下可能都有子目录“ project1”。如果光是对节点名“ project1”杂凑，则势必至少有上百个 dentry 结构都挂在同一队列中而需要线性搜索。即使把父节点名也一起杂凑还是解决不了问题，因为每个学生都可能会有例如“ project1/src”，所有此类路径中的“ src”节点又会在同一个队列中，对全路径名进行杂凑当然可以解决问题，但是那样代价

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

又太高了。

找到了相应的队列头部以后，d_lookup()中的 for 循环是简单的。惟一特殊之处是具体的文件系统可能通过其 dentry_operations 结构提供自己的节点名比对函数（比方说，有些文件系统可能在比对时跳过所有的空格），没有的话就用普通的 memcmp()。

回到 cached_lookup()的代码中，具体的文件系统可能通过其 dentry_operations 结构提供一个对找到的 dentry 结构进行验证（和处理）的函数，如果验证失败就要通过 d_invalidate()将这个数据结构从杂凑队列中脱链，这种安排对有些文件系统是必要的，例如在“网络文件系统”NFS 中，如果一个远程的进程是其惟一的用户，又有很长时间没有访问这个结构了，那就应该将其视作无效，而根据磁盘上的父目录内容来重新构造。具体的函数由该文件系统的 dentry_operations 结构中通过函数指针 d_revalidate 提供，最后则根据验证的结果返回一个 dentry 指针或出错代码。不过，有的文件系统根本就不提供 dentry_operations 数据结构，所以其 dentry 结构中的 d_op 是 0，表示按 Linux 默认的方式处理各项目录项操作。事实上，Ext2 就不提供其自己的 dentry_operations 结构，因为 Linux 默认的方式就是 Ext2。进一步，即使某个文件系统提供了自己的 dentry_operations 数据结构，也并不一定提供自己的 d_revalidate 操作。所以，代码中要先对这两个指针加以检验。由于 Ext2 并不提供其自己的 dentry_operations 结构，我们就把它跳过了。

至此，cache_lookup()就完成了。

如果所需的 dentry 结构不在杂凑表队列中或者已经无效，则返回 NULL。那样，就要进一步通过 real_lookup()从父目录在磁盘上的内容中找到本节点的目录项，再根据其内容在内存中为之建立起一个 dentry 结构（见 path_walk()的 497 行）。下面就是 real_lookup()的代码（见 fs/namei.c）：

==================== fs/namei.c 260 310 ====================

[path_walk()>real_lookup()]

```
/*
 * This is called when everything else fails, and we actually have
 * to go to the low-level filesystem to find out what we should do..
 *
 * We get the directory semaphore, and after getting that we also
 * make sure that nobody added the entry to the dcache in the meantime..
 * SMP-safe
 */
static struct dentry * real_lookup(struct dentry * parent, struct qstr * name, int flags)
{
     struct dentry * result;
     struct inode *dir = parent->d_inode;

     down(&dir->i_sem);
     /*
      * First re-do the cached lookup just in case it was created
      * while we waited for the directory semaphore..
      *
      * FIXME! This could use version numbering or similar to
      * avoid unnecessary cache lookups.
      */
     result = d_lookup(parent, name);
```

```
282         if (!result) {
283             struct dentry * dentry = d_alloc(parent, name);
284             result = ERR_PTR(-ENOMEM);
285             if (dentry) {
286                 lock_kernel();
287                 result = dir->i_op->lookup(dir, dentry);
288                 unlock_kernel();
289                 if (result)
290                     dput(dentry);
291                 else
292                     result = dentry;
293             }
294             up(&dir->i_sem);
295             return result;
296         }
297 
298         /*
299          * Uhhuh! Nasty case: the cache was re-populated while
300          * we waited on the semaphore. Need to revalidate.
301          */
302         up(&dir->i_sem);
303         if (result->d_op && result->d_op->d_revalidate) {
304             if (!result->d_op->d_revalidate(result, flags) && !d_invalidate(result)) {
305                 dput(result);
306                 result = ERR_PTR(-ENOENT);
307             }
308         }
309         return result;
310     }
```

建立 dentry 结构的过程不容许受到其它进程的干扰，所以必须通过信号量放在临界区中进行。但是，在通过 down()进入临界区时可能会经历一段睡眠等待的时间，而其它进程有可能已经在这段时间中把所需的 dentry 结构建立好，再建立一个就重复了。所以，在进入临界区以后，还要再通过 d_lookup()确认一下所需的 dentry 结构确实不在杂凑表队列中。读者在前面几章中也看到过类似的情况，总的来说，这是一种规范性的处理方式。万一真的发生了这种情况，那就根据具体文件系统的要求而（可能）调用一个函数进行一些验证和处理（与 cached_lookup()中相似）。当然，发生这种情况的概率是很低的，在多数情况下都需要建立 dentry 结构。

要建立起一个 dentry 结构，首先当然要为之分配空间并初始化，这是由 283 行的 d_alloc()完成的，其代码在 fs/dcache.c 中：

==================== fs/dcache.c 589 646 ====================

[path_walk()>real_lookup()>d_alloc()]

```
/**
 * d_alloc      -     allocate a dcache entry
 * @parent: parent of entry to allocate
 * @name: qstr of the name
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Allocates a dentry. It returns %NULL if there is insufficient memory
 * available. On a success the dentry is returned. The name passed in is
 * copied and the copy passed in may be reused after this call.
 */

struct dentry * d_alloc(struct dentry * parent, const struct qstr *name)
{
    char * str;
    struct dentry *dentry;

    dentry = kmem_cache_alloc(dentry_cache, GFP_KERNEL);
    if (!dentry)
        return NULL;

    if (name->len > DNAME_INLINE_LEN-1) {
        str = kmalloc(NAME_ALLOC_LEN(name->len), GFP_KERNEL);
        if (!str) {
            kmem_cache_free(dentry_cache, dentry);
            return NULL;
        }
    } else
        str = dentry->d_iname;

    memcpy(str, name->name, name->len);
    str[name->len] = 0;

    atomic_set(&dentry->d_count, 1);
    dentry->d_flags = 0;
    dentry->d_inode = NULL;
    dentry->d_parent = NULL;
    dentry->d_sb = NULL;
    dentry->d_name.name = str;
    dentry->d_name.len = name->len;
    dentry->d_name.hash = name->hash;
    dentry->d_op = NULL;
    dentry->d_fsdata = NULL;
    INIT_LIST_HEAD(&dentry->d_vfsmnt);
    INIT_LIST_HEAD(&dentry->d_hash);
    INIT_LIST_HEAD(&dentry->d_lru);
    INIT_LIST_HEAD(&dentry->d_subdirs);
    INIT_LIST_HEAD(&dentry->d_alias);
    if (parent) {
        dentry->d_parent = dget(parent);
        dentry->d_sb = parent->d_sb;
        spin_lock(&dcache_lock);
        list_add(&dentry->d_child, &parent->d_subdirs);
        spin_unlock(&dcache_lock);
    } else
        INIT_LIST_HEAD(&dentry->d_child);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

	dentry_stat.nr_dentry++;
	return dentry;
}
```

从这段程序中我们可以看到，dentry 数据结构是通过 kmem_alloc()从为这种数据结构专设的 slab 队列中分配的。当节点名较短时，dentry 结构中有个字符数组 d_iname 用来保存节点名，不然就要另行为之分配空间。不管怎样，dentry 结构中的 d_name.name 总是指向这个字符串。此外，dentry 结构中指向超级块结构的指针 d_sb 是从其父节点（目录）继承下来的。每当建立了一个 dentry 结构时，就要将其父节点（“ /” 除外，它没有父节点）的共享计数通过 dget()递增，所以这个新建的 dentry 结构就成了其父节点的 dentry 结构的一个“用户”，并且要挂入父节点的 d_subdirs 队列中。注意父节点的 d_subdirs 队列中只包含在内存中建有 dentry 结构的目录项。

回到 real_lookup()的代码中。分配了空间以后，就要从磁盘上由父节点代表的那个目录中寻找当前节点的目录项并设置结构中的其它信息。如果寻找失败，就通过 dput()撤销已经分配空间的 dentry 结构。如果成功，就通过函数 real_lookup()的 295 行的 return 语句返回指向该 dentry 结构的指针。

从磁盘上寻找的过程因文件系统而异，所以要通过父节点 inode 结构中的指针 i_op 找到相应的 inode_operations 数据结构。对于代表着目录的 inode 和代表着文件的 inode，其 inode_operation 结构常常是不同的。就 Ext2 而言，对于目录节点的函数跳转结构为 ext2_dir_inode_operations，定义见 fs/ext2/namei.c：

==================== fs/ext2/namei.c 811 824 ====================

```
/*
 * directories can handle most operations...
 */
struct inode_operations ext2_dir_inode_operations = {
	create:		ext2_create,
	lookup:		ext2_lookup,
	link:		ext2_link,
	unlink:		ext2_unlink,
	symlink:	ext2_symlink,
	mkdir:		ext2_mkdir,
	rmdir:		ext2_rmdir,
	mknod:		ext2_mknod,
	rename:		ext2_rename,
};
```

可见，具体的函数为 ext2_lookup()，其代码在同一文件（fs/ext2/namei.c）中：

==================== fs/ext2/namei.c 163 184 ====================

[path_walk()>real_lookup()>ext2_lookup()]

```
static struct dentry *ext2_lookup(struct inode * dir, struct dentry *dentry)
{
	struct inode * inode;
	struct ext2_dir_entry_2 * de;
	struct buffer_head * bh;

```

```
    if (dentry->d_name.len > EXT2_NAME_LEN)
        return ERR_PTR(-ENAMETOOLONG);

    bh = ext2_find_entry (dir, dentry->d_name.name, dentry->d_name.len, &de);
    inode = NULL;
    if (bh) {
        unsigned long ino = le32_to_cpu(de->inode);
        brelse (bh);
        inode = iget(dir->i_sb, ino);

        if (!inode)
            return ERR_PTR(-EACCES);
    }
    d_add(dentry, inode);
    return NULL;
}
```

这里先由 ext2_find_entry()从磁盘上找到并读入当前节点的目录项，然后通过 iget()根据索引节点号从磁盘读入相应索引节点并在内存中建立起相对的 inode 结构，最后，由 d_add 完成 dentry 结构的设置并将其挂入杂凑表中的某个队列。

函数 ext2_find_entry()的代码也在 fs/ext2/namei.c 中：

==================== fs/ext2/namei.c 52 161 ====================

[path_walk()>real_lookup()>ext2_lookup()>ext2_find_entry()]

```
/*
 *   ext2_find_entry()
 *
 * finds an entry in the specified directory with the wanted name. It
 * returns the cache buffer in which the entry was found, and the entry
 * itself (as a parameter - res_dir). It does NOT read the inode of the
 * entry - you'll have to do that yourself if you want to.
 */
static struct buffer_head * ext2_find_entry (struct inode * dir,
                             const char * const name, int namelen,
                             struct ext2_dir_entry_2 ** res_dir)
{
    struct super_block * sb;
    struct buffer_head * bh_use[NAMEI_RA_SIZE];
    struct buffer_head * bh_read[NAMEI_RA_SIZE];
    unsigned long offset;
    int block, toread, i, err;

    *res_dir = NULL;
    sb = dir->i_sb;

    if (namelen > EXT2_NAME_LEN)
        return NULL;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    memset (bh_use, 0, sizeof (bh_use));
    toread = 0;
    for (block = 0; block < NAMEI_RA_SIZE; ++block) {
        struct buffer_head * bh;

        if ((block << EXT2_BLOCK_SIZE_BITS (sb)) >= dir->i_size)
            break;
        bh = ext2_getblk (dir, block, 0, &err);
        bh_use[block] = bh;
        if (bh && !buffer_uptodate(bh))
            bh_read[toread++] = bh;
    }

    for (block = 0, offset = 0; offset < dir->i_size; block++) {
        struct buffer_head * bh;
        struct ext2_dir_entry_2 * de;
        char * dlimit;

        if ((block % NAMEI_RA_BLOCKS) == 0 && toread) {
            ll_rw_block (READ, toread, bh_read);
            toread = 0;
        }
        bh = bh_use[block % NAMEI_RA_SIZE];
        if (!bh) {
#if 0
            ext2_error (sb, "ext2_find_entry",
                    "directory #%lu contains a hole at offset %lu",
                    dir->i_ino, offset);
#endif
            offset += sb->s_blocksize;
            continue;
        }
        wait_on_buffer (bh);
        if (!buffer_uptodate(bh)) {
            /*
             * read error: all bets are off
             */
            break;
        }

        de = (struct ext2_dir_entry_2 *) bh->b_data;
        dlimit = bh->b_data + sb->s_blocksize;
        while ((char *) de < dlimit) {
            /* this code is executed quadratically often */
            /* do minimal checking `by hand' */
            int de_len;

            if ((char *) de + namelen <= dlimit &&
                ext2_match (namelen, name, de)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                /* found a match -
                   just to be sure, do a full check */
                if (!ext2_check_dir_entry("ext2_find_entry",
                                          dir, de, bh, offset))
                    goto failure;
                for (i = 0; i < NAMEI_RA_SIZE; ++i) {
                    if (bh_use[i] != bh)
                        brelse (bh_use[i]);
                }
                *res_dir = de;
                return bh;
            }
            /* prevent looping on a bad block */
            de_len = le16_to_cpu(de->rec_len);
            if (de_len <= 0)
                goto failure;
            offset += de_len;
            de = (struct ext2_dir_entry_2 *)
                ((char *) de + de_len);
        }

        brelse (bh);
        if (((block + NAMEI_RA_SIZE) << EXT2_BLOCK_SIZE_BITS (sb)) >=
           dir->i_size)
            bh = NULL;
        else
            bh = ext2_getblk (dir, block + NAMEI_RA_SIZE, 0, &err);
        bh_use[block % NAMEI_RA_SIZE] = bh;
        if (bh && !buffer_uptodate(bh))
            bh_read[toread++] = bh;
    }

failure:
    for (i = 0; i < NAMEI_RA_SIZE; ++i)
        brelse (bh_use[i]);
    return NULL;
}
```

这段程序涉及文件的读操作，读者可以在学习了“文件的读写”一节以及本书下册“设备驱动”一章后再回过来仔细阅读，这里只作一些必要的说明。目录其实只是一种特殊格式的文件，就 Ext2 文件系统而言，目录文件的内容在概念上就是一个 ext2_dir_entry_2 结构数组（见前节），其目的仅在于根据节点名（最长可达 255 个字符）得到相应的索引节点号，所以从逻辑的角度讲是很简单的。为什么只是说“概念上”是 ext2_dir_entry_2 结构数组呢？因为实际上不是。前一节中讲过，ext2_dir_entry_2 结构的长度是不固定的（节点名可长达 255，但通常只是几个字符，而一个文件系统中也许有数万个目录项），结构中有个字段 rec_len 指明本结构的长度。既然不是固定长度的，就不能像真正的数组那样通过下标来计算出具体元素的位置，而只好采用线性搜索的办法（这是一个“以时间换空间”的例子）。不过，为了避免因目录项跨磁盘记录块而造成处理上的不便，Ext2 文件系统在为目录项分配磁盘空间

PDF-XCHANGE Click to buy NOW! www.docu-track.com

时不让跨记录块。如果一个记录块中剩下的空间已经不够就另起一个记录块。

不同的处理器在存取数据时在字节的排列次序上有所谓“ big ending”和“ little ending”之分。例如，i386 就是“ little ending”的处理器，它在存储一个 16 位数据 0x1234 时实际存储的却是 0x3412，对于 32 位数据也与此类似。这里的索引节点号与记录长度都作为 32 位或 16 位无符号整数存储在磁盘上的，而同一磁盘既可以安装在采用“ little ending”方式的 CPU 的机器上，也可能安装在采用“ big ending ”方式的 CPU 的机器上，所以要选择一种形式作为标准。事实上，Ext2 采用的标准是“ little ending”，所以在使用存储在磁盘上大于 8 位的整数时要先通过 le32_to_cpu()、lel6_to_cpu()等函数将这些数据从“ little ending”形式转换成具体 CPU 所采用的形式。当然，在 i386 处理器上访问 Ext2 文件系统时这这些函数实际上不作任何转换。

由于磁盘的物理特性，从磁盘读一个记录块需要一定的时间，而这些时间主要消耗在准备工作上。一旦准备好了，读一个记录块与读几个记录块所需时间其实相差不大。所以比较好的办法是既然读了就往前“预读”（Read Ahead）一些记录块，因为紧接着的这些记录块很可能马上就要用到。另一方面，从磁盘读记录块的操作一经启动便由磁盘自行完成，而无需 CPU 介入。所以，从读的第一批记录块到位以后，CPU 对记录块的处理就可以跟后续记录块的读入相平行，从而形成流水操作。那么往前多少块比较合适呢？那要看具体情况了，对于从磁盘读入目录内容这个特定的目的，代码中定义了几个常数（见文件 fs/ext2/namei.c）：

==================== fs/ext2/namei.c 28 34 ====================

```
/*
 * define how far ahead to read directories while searching them.
 */
#define NAMEI_RA_CHUNKS  2
#define NAMEI_RA_BLOCKS  4
#define NAMEI_RA_SIZE       (NAMEI_RA_CHUNKS * NAMEI_RA_BLOCKS)
#define NAMEI_RA_INDEX(c,b)  (((c) * NAMEI_RA_BLOCKS) + (b))
```

所以，预读的提前量为 8 个记录块，也就是说，估计在读入一个记录块所需的时间内 CPU 可以处理 8 个记录块。

对于从磁盘读入的记录块都要在前面加上一个头部，即 buffer_head 数据结构以便管理。由于从磁盘读入一个记录块的代价不小，对已经读入的记录块都不是用了即扔的，而是要在内存中加以缓冲存储。所以，有时候并不需要真的到磁盘上去读。但是，这样一来有时候缓冲存储着的记录块与磁盘上的记录块就可能不一致了。

代码中为记录块设置了两个指针数组，一个是 bh_use[]，另一个是 bh_read[]，大小都是 NAMEI_RA_SIZE，即 8。首先通过一个 for 循环，调用 ext2_getblk()从缓冲着的记录块中找到给定目录文件的开头 8 个逻辑记录块，或者就为之分配缓冲区，并将它们的 buffer_head 结构指针写入数组 bh_use[]，将 bh_use[]填满。这就是要搜索的第一批次。当然，如果这个目录文件的大小还不够 8 个记录块（见 78 行）那又另作别论（注意，参数 dir 指向其 inode 结构，而不是 dentry 结构）。在这 8 个记录块中，如果有的已经与磁盘上不一致（见 85 行），则要在另一个数组 bh_read[]中记录下来，这就是真正要从磁盘上读的。至于新分配的缓冲区，那当然与磁盘上不一致。

接着是对目录文件中所有记录块的 for 循环，对目标节点的搜索就是扫描所有记录块中的所有目录

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

项。循环从 block 0 开始，每隔 NAMEI_RA_BLOCKS 个就启动一次读磁盘操作（如果需要的话），每次最多读 8 块，而数组 bh_read[]则给出所需记录块的“名单”。第一次把 8 个缓冲区填满以后，再往后的从磁盘读入与 CPU 的处理就可以形成一种流水线式的操作了。由于从磁盘读入是异步的，CPU 在每处理一个记录块之前都要通过 wait_on_buffer()等待该记录块到位。但是只要预读的参数合适就可以达到基本上不需要什么等待的程度。

至于在记录块中搜索的过程，那就很简单了（见 116～144 行）。虽然 Ext2 的目录项是可变大小的，但是却不会跨记录块存储，所以每个记录块的开始必然也是一个目录项的开始（见 116 行），而一个记录块内有几个目录项那就不一定了。在找到了所需的目录项以后，要将其它的记录块缓冲区释放，只留下该目录项所在的那个记录块（见 130～133 行）。最后返回目录项所在的记录块，并通过参数 res_dir 返回目录项指针。

回到 ext2_lookup()的代码中，下一步是根据查得的索引节点号通过 iget()找到或建立起所需的 inode 结构，这里的 iget()是个 inline 函数，定义于 indude/linux/fs.h：

==================== include/linux/fs.h 1185 1188 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()]

```
static inline struct inode *iget(struct super_block *sb, unsigned long ino)
{
	return iget4(sb, ino, NULL, NULL);
}
```

函数 iget4()的代码在 fs/inode.c 中：

==================== fs/inode.c 774 794 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()>iget4()]

```
struct inode *iget4(struct super_block *sb, unsigned long ino, find_inode_t find_actor, void *opaque)
{
	struct list_head * head = inode_hashtable + hash(sb,ino);
	struct inode * inode;

	spin_lock(&inode_lock);
	inode = find_inode(sb, ino, head, find_actor, opaque);
	if (inode) {
		__iget(inode);
		spin_unlock(&inode_lock);
		wait_on_inode(inode);
		return inode;
	}
	spin_unlock(&inode_lock);

	/*
	 * get_new_inode() will do the right thing, re-trying the search
	 * in case it had to block at any point.
	 */
	return get_new_inode(sb, ino, head, find_actor, opaque);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

同样，目标节点的 inode 结构也可能已经在内存中，也可能需要从磁盘上读入其索引节点后在内存中创建。就像 dentry 结构有个杂凑表 dentry_hashtable 一样， inode 结构也有个杂凑表 inode_hashtable，已经建立的 inode 结构都通过结构中的 i_hash（也是一个 list_head）挂在该杂凑表的某一个队列中，所以首先要通过 find_inode()在杂凑表队列中寻找。找到后就通过 iget()递增其共享计数。由于索引节点号只在同一设备上才是惟一的，在杂凑计算时要把所在设备的 super_block 结构的地址也结合进去。

要是杂凑表的队列中找不到所需的 inode 结构，那就要通过 get_new_inode()从磁盘上读入相应的索引节点并建立起一个 inode 结构，其代码在 fs/inode.c 中：

==================== fs/inode.c 649 707 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()]

```
/*
 * This is called without the inode lock held.. Be careful.
 *
 * We no longer cache the sb_flags in i_flags - see fs.h
 *   -- rmk@arm.uk.linux.org
 */
static struct inode * get_new_inode(struct super_block *sb, unsigned long ino, struct list_head *head, find_inode_t find_actor, void *opaque)
{
    struct inode * inode;

    inode = alloc_inode();
    if (inode) {
        struct inode * old;

        spin_lock(&inode_lock);
        /* We released the lock, so.. */
        old = find_inode(sb, ino, head, find_actor, opaque);
        if (!old) {
            inodes_stat.nr_inodes++;
            list_add(&inode->i_list, &inode_in_use);
            list_add(&inode->i_hash, head);
            inode->i_sb = sb;
            inode->i_dev = sb->s_dev;
            inode->i_ino = ino;
            inode->i_flags = 0;
            atomic_set(&inode->i_count, 1);
            inode->i_state = I_LOCK;
            spin_unlock(&inode_lock);

            clean_inode(inode);
            sb->s_op->read_inode(inode);

            /*
             * This is special!  We do not need the spinlock
             * when clearing I_LOCK, because we're guaranteed
             * that nobody else tries to do anything about the
```

```
                 * state of the inode when it is locked, as we
                 * just created it (so there can be no old holders
                 * that haven't tested I_LOCK).
                 */
                inode->i_state &= ~I_LOCK;
                wake_up(&inode->i_wait);

                return inode;
            }

            /*
             * Uhhuh, somebody else created the same inode under
             * us. Use the old inode instead of the one we just
             * allocated.
             */
            __iget(old);
            spin_unlock(&inode_lock);
            destroy_inode(inode);
            inode = old;
            wait_on_inode(inode);
        }
        return inode;
    }
```

这个函数的代码与前面 dentry 结构的分配和建立很相似，至于从磁盘读入索引节点的过程则取决于具体的文件系统。有的文件系统在磁盘上可能并不存在“索引节点”这么一种东西，但是在概念上必有相通之处。所以，对有的文件系统来说是“读入”索引节点，而对有的文件系统则是将磁盘上的有关信息变换成一个索引节点。其实，所谓超级块、目录项也莫不如此。注意代码中对新创建 inode 结构中 i_dev 等字段的设置。以前讲过，inode 结构中的 i_dev 表示这个结构所代表的文件所在的设备，这里可以看到它的值来自所在设备的 super_block 数据结构，是在从设备上读入索引节点之前就设置好了的。还有，索引节点号 ino 仅在同一设备上才是惟一的，所以要与设备号（或 super_block 结构）合在一起才能在全系统范围中惟一地确定一个索引节点及其 inode 结构。这也是为什么 find_inode()的参数表中包括了 sb 和 ino 的原因。

对于索引节点的读入，具体的函数是通过函数跳转表 super_operations 结构中的函数指针 read_inode 提供的。每个设备的 super_block 结构中都有一个指针 s_op，指向具体的跳转表。对于 Ext2 来说，这个跳转表就是 ext2_sops，具体的函数则是 ext2_read_inode()（见文件 fs/ext2/super.c）。

==================== fs/ext2/super.c 148 157 ====================

```
static struct super_operations ext2_sops = {
    read_inode:   ext2_read_inode,
    write_inode:  ext2_write_inode,
    put_inode:    ext2_put_inode,
    delete_inode: ext2_delete_inode,
    put_super:    ext2_put_super,
    write_super:  ext2_write_super,
    statfs:       ext2_statfs,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        remount_fs:   ext2_remount,
};
```

函数 ext2_read_inode()的代码则在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 961 1057 ====================
[path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()>ext2_read_inode()]

```
void ext2_read_inode (struct inode * inode)
{
        struct buffer_head * bh;
        struct ext2_inode * raw_inode;
        unsigned long block_group;
        unsigned long group_desc;
        unsigned long desc;
        unsigned long block;
        unsigned long offset;
        struct ext2_group_desc * gdp;

        if ((inode->i_ino != EXT2_ROOT_INO && inode->i_ino != EXT2_ACL_IDX_INO &&
             inode->i_ino != EXT2_ACL_DATA_INO &&
             inode->i_ino < EXT2_FIRST_INO(inode->i_sb)) ||
            inode->i_ino > le32_to_cpu(inode->i_sb->u.ext2_sb.s_es->s_inodes_count)) {
             ext2_error (inode->i_sb, "ext2_read_inode",
                         "bad inode number: %lu", inode->i_ino);
             goto bad_inode;
        }
        block_group = (inode->i_ino - 1) / EXT2_INODES_PER_GROUP(inode->i_sb);
        if (block_group >= inode->i_sb->u.ext2_sb.s_groups_count) {
             ext2_error (inode->i_sb, "ext2_read_inode",
                         "group >= groups count");
             goto bad_inode;
        }
        group_desc = block_group >> EXT2_DESC_PER_BLOCK_BITS(inode->i_sb);
        desc = block_group & (EXT2_DESC_PER_BLOCK(inode->i_sb) - 1);
        bh = inode->i_sb->u.ext2_sb.s_group_desc[group_desc];
        if (!bh) {
             ext2_error (inode->i_sb, "ext2_read_inode",
                         "Descriptor not loaded");
             goto bad_inode;
        }

        gdp = (struct ext2_group_desc *) bh->b_data;
        /*
         * Figure out the offset within the block group inode table
         */
        offset = ((inode->i_ino - 1) % EXT2_INODES_PER_GROUP(inode->i_sb)) *
             EXT2_INODE_SIZE(inode->i_sb);
        block = le32_to_cpu(gdp[desc].bg_inode_table) +
             (offset >> EXT2_BLOCK_SIZE_BITS(inode->i_sb));
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	if (!(bh = bread (inode->i_dev, block, inode->i_sb->s_blocksize))) {
		ext2_error (inode->i_sb, "ext2_read_inode",
			    "unable to read inode block - "
			    "inode=%lu, block=%lu", inode->i_ino, block);
		goto bad_inode;
	}
	offset &= (EXT2_BLOCK_SIZE(inode->i_sb) - 1);
	raw_inode = (struct ext2_inode *) (bh->b_data + offset);

	inode->i_mode = le16_to_cpu(raw_inode->i_mode);
	inode->i_uid = (uid_t)le16_to_cpu(raw_inode->i_uid_low);
	inode->i_gid = (gid_t)le16_to_cpu(raw_inode->i_gid_low);
	if(!(test_opt (inode->i_sb, NO_UID32))) {
		inode->i_uid |= le16_to_cpu(raw_inode->i_uid_high) << 16;
		inode->i_gid |= le16_to_cpu(raw_inode->i_gid_high) << 16;
	}
	inode->i_nlink = le16_to_cpu(raw_inode->i_links_count);
	inode->i_size = le32_to_cpu(raw_inode->i_size);
	inode->i_atime = le32_to_cpu(raw_inode->i_atime);
	inode->i_ctime = le32_to_cpu(raw_inode->i_ctime);
	inode->i_mtime = le32_to_cpu(raw_inode->i_mtime);
	inode->u.ext2_i.i_dtime = le32_to_cpu(raw_inode->i_dtime);
	/* We now have enough fields to check if the inode was active or not.
	 * This is needed because nfsd might try to access dead inodes
	 * the test is that same one that e2fsck uses
	 * NeilBrown 1999oct15
	 */
	if (inode->i_nlink == 0 && (inode->i_mode == 0 || inode->u.ext2_i.i_dtime)) {
		/* this inode is deleted */
		brelse (bh);
		goto bad_inode;
	}
	inode->i_blksize = PAGE_SIZE;	/* This is the optimal IO size (for stat), not the fs block size */
	inode->i_blocks = le32_to_cpu(raw_inode->i_blocks);
	inode->i_version = ++event;
	inode->u.ext2_i.i_flags = le32_to_cpu(raw_inode->i_flags);
	inode->u.ext2_i.i_faddr = le32_to_cpu(raw_inode->i_faddr);
	inode->u.ext2_i.i_frag_no = raw_inode->i_frag;
	inode->u.ext2_i.i_frag_size = raw_inode->i_fsize;
	inode->u.ext2_i.i_file_acl = le32_to_cpu(raw_inode->i_file_acl);
	if (S_ISDIR(inode->i_mode))
		inode->u.ext2_i.i_dir_acl = le32_to_cpu(raw_inode->i_dir_acl);
	else {
		inode->u.ext2_i.i_high_size = le32_to_cpu(raw_inode->i_size_high);
		inode->i_size |= ((__u64)le32_to_cpu(raw_inode->i_size_high)) << 32;
	}
	inode->i_generation = le32_to_cpu(raw_inode->i_generation);
	inode->u.ext2_i.i_block_group = block_group;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1051
1052     /*
1053      * NOTE! The in-memory inode i_data array is in little-endian order
1054      * even on big-endian machines: we do NOT byteswap the block numbers!
1055      */
1056     for (block = 0; block < EXT2_N_BLOCKS; block++)
1057          inode->u.ext2_i.i_data[block] = raw_inode->i_block[block];
```

在 Ext2 格式的磁盘上，有些索引节点是有特殊用途的，include/linux/ext2_fs.h 中有这些节点的定义：

==================== include/linux/ext2_fs.h 55 63 ====================

```
/*
 * Special inodes numbers
 */
#define  EXT2_BAD_INO         1   /* Bad blocks inode */
#define EXT2_ROOT_INO         2   /* Root inode */
#define EXT2_ACL_IDX_INO      3   /* ACL inode */
#define EXT2_ACL_DATA_INO     4   /* ACL inode */
#define EXT2_BOOT_LOADER_INO      5   /* Boot loader inode */
#define EXT2_UNDEL_DIR_INO    6   /* Undelete directory inode */
```

这些索引节点是为系统保留的，对它们的访问都不通过目录项而直接通过定义的节点号进行。其中 EXT2_ACL_IDX_INO 和 EXT2_ACL_DATA_INO 用于"访问控制表"（access control list），是为改善文件系统的安全性而设置的（见"访问权限和文件的安全性"一节）。磁盘设备的 super_block 结构中提供磁盘上第一个供常规用途的索引节点的节点号以及索引节点的总数，这两项参数被用于对节点号的范围检查。

从概念上说，Ext2 格式的磁盘设备上，除引导块和超级块以外，就分成索引节点和数据两部分。但是，出于访问效率的考虑，实际上把整个磁盘（或逻辑磁盘，即"分区"）先划分成若干"记录块组"，然后再将每个记录块组分成索引节点和数据两部分。与此相应，ext2 磁盘的超级块中则提供有关这种划分的参数，如磁盘上有多少个组，每个组中有多少个记录块，有多少个索引节点等等；同时，每个块组还有一个"组描述结构"，也可以通过 super_block 结构访问（详见"文件系统的安装与拆卸"）。所以，先要根据索引节点号算出该节点所在的记录块组（见 980 行）以及在节点组内的位移（999 行），然后再算出节点所在的记录块号（1001 行）。知道了记录块号以后，就可以通过设备驱动程序 bread 读入该记录块。从磁盘读入的索引节点为 ext2_inode 数据结构，读者已经看到过它的定义。索引节点中的信息是原始的，未经过加工的，所以代码中称之为 raw_inode，相比之下内存中 inode 结构中的信息则分两个部分，一部分是属于 VFS 层的，适用于所有的文件系统；另一部分则属于具体的文件系统，这就是那个 union，因具体文件系统的不同而赋予不同的解释。对 Ext2 来说，这部分数据形成一个 ext2_inode_info 结构，这是在 include/linux/ext2_fs.h 中定义的：

==================== include/linux/ext2_fs_i.h 19 40 ====================

```
/*
 * second extended file system inode data in memory
 */
struct ext2_inode_info {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    __u32   i_data[15];
    __u32   i_flags;
    __u32   i_faddr;
    __u8 i_frag_no;
    __u8 i_frag_size;
    __u16   i_osync;
    __u32   i_file_acl;
    __u32   i_dir_acl;
    __u32   i_dtime;
    __u32   not_used_1;   /* FIX: not used/ 2.2 placeholder */
    __u32   i_block_group;
    __u32   i_next_alloc_block;
    __u32   i_next_alloc_goal;
    __u32   i_prealloc_block;
    __u32   i_prealloc_count;
    __u32   i_high_size;
    int  i_new_inode:1;     /* Is a freshly allocated inode */
};
```

结构中的 i_data[]是一块很重要的数据。对于有存储内容的文件（普通文件和目录文件），这里存放着一些指针，直接或间接地指向磁盘上存储着该文件内容的所有记录块（详见“文件的读与写”一节）。所谓“索引节点”即因此而得名。至于代表着符号连接的节点，则并没有文件内容（数据），所以正好用这块空间来存储连接目标的路径名。这块空间的大小是 15 个 32 位整数，即 60 个字节。虽然节点名最长可达 255 个字节，但一般都不会很长，将作为符号连接目标的路径名限制在 60 个字节不至于引起问题。代码中通过一个 for 循环将这 15 个整数复制到 inode 结构的 union 中。

在 ext2_read_inode()的代码中继续往下看（fs/ext2/inode.c）：

==================== fs/ext2/inode.c 1059 1102 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()>ext2_read_inode()]

```
    if (inode->i_ino == EXT2_ACL_IDX_INO ||
       inode->i_ino == EXT2_ACL_DATA_INO)
        /* Nothing to do */ ;
    else if (S_ISREG(inode->i_mode)) {
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        inode->i_mapping->a_ops = &ext2_aops;
    } else if (S_ISDIR(inode->i_mode)) {
        inode->i_op = &ext2_dir_inode_operations;
        inode->i_fop = &ext2_dir_operations;
    } else if (S_ISLNK(inode->i_mode)) {
        if (!inode->i_blocks)
            inode->i_op = &ext2_fast_symlink_inode_operations;
        else {
            inode->i_op = &page_symlink_inode_operations;
            inode->i_mapping->a_ops = &ext2_aops;
        }
    } else
        init_special_inode(inode, inode->i_mode,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    le32_to_cpu(raw_inode->i_block[0]));
    brelse (bh);
    inode->i_attr_flags = 0;
    if (inode->u.ext2_i.i_flags & EXT2_SYNC_FL) {
        inode->i_attr_flags |= ATTR_FLAG_SYNCRONOUS;
        inode->i_flags |= S_SYNC;
    }
    if (inode->u.ext2_i.i_flags & EXT2_APPEND_FL) {
        inode->i_attr_flags |= ATTR_FLAG_APPEND;
        inode->i_flags |= S_APPEND;
    }
    if (inode->u.ext2_i.i_flags & EXT2_IMMUTABLE_FL) {
        inode->i_attr_flags |= ATTR_FLAG_IMMUTABLE;
        inode->i_flags |= S_IMMUTABLE;
    }
    if (inode->u.ext2_i.i_flags & EXT2_NOATIME_FL) {
        inode->i_attr_flags |= ATTR_FLAG_NOATIME;
        inode->i_flags |= S_NOATIME;
    }
    return;

bad_inode:
    make_bad_inode(inode);
    return;
}
```

接着，就是根据由索引节点所提供的信息设置 inode 结构中的 inode_operations 结构指针和 file_operations 结构指针，完成具体文件系统与虚拟文件系统 VFS 之间的连接。以前，我们曾把这二者比喻成“接口卡”与“总线”，但是读者要注意这是从系统结构的角度而言的，实际上文件系统中的每个节点（目录或文件）都有从 VFS 层连接到具体文件系统的问题，就好像每个节点都有着这么一条“总线”一样。

目前的 2.4 版 Linux 内核并不支持 ACL（Access Control List），所以代码中只是为之留下了位置，而暂时不作任何处理。除此以外，就通过检查 inode 结构中的 inode 字段来确定该节点是否常规文件（S_ISREG）、目录（S_ISDIR）、符号连接（S_ISLNK）或特殊文件而作不同的设置或处理。例如对（Ext2 文件系统的）目录节点就将 i_op 和 i_fop 分别设置成指向 ext2_dir_inode_operations 和 ext2_dir_operations。对于常规文件则除 i_op 和 i_fop 以外还有另一个指针 a_ops，它指向一个 address_space_operations 数据结构，用于文件到内存空间的映射或缓冲。对特殊文件则通过 init_special_inode()加以检查和处理，以后我们将常常回过来看这个函数。

在找到了或者建立了所需的 inode 结构以后，就返回到 ext2_lookup()，在那里还要通过 d_add()将 inode 结构与 dentry 结构挂上钩，并将 dentry 结构挂入杂凑表中的某个队列。这里的 d_add()是个 inline 函数，定义于 include/linux/dcache.h：

==================== include/linux/dcache.h 191 204 ====================
[path_walk()>real_lookup()>ext2_lookup()>d_add()]

```
/**
```

```
 * d_add - add dentry to hash queues
 * @entry: dentry to add
 * @inode: The inode to attach to this dentry
 *
 * This adds the entry to the hash queues and initializes @inode.
 * The entry was actually filled in earlier during d_alloc().
 */

static __inline__ void d_add(struct dentry * entry, struct inode * inode)
{
     d_instantiate(entry, inode);
     d_rehash(entry);
}
```

函数 d_instantiate()使 dentry 结构和 inode 结构互相挂钩，其代码在 fs/dcache.c 中：

==================== fs/dcache.c 648 670 ====================

[path_walk()>real_lookup()>ext2_lookup()>d_add()>d_instantiate()]

```
/**
 * d_instantiate - fill in inode information for a dentry
 * @entry: dentry to complete
 * @inode: inode to attach to this dentry
 *
 * Fill in inode information in the entry.
 *
 * This turns negative dentries into productive full members
 * of society.
 *
 * NOTE! This assumes that the inode count has been incremented
 * (or otherwise set) by the caller to indicate that it is now
 * in use by the dcache.
 */

void d_instantiate(struct dentry *entry, struct inode * inode)
{
     spin_lock(&dcache_lock);
     if (inode)
          list_add(&entry->d_alias, &inode->i_dentry);
     entry->d_inode = inode;
     spin_unlock(&dcache_lock);
}
```

两个数据结构之间的联系是双向的。一方面是 dentry 结构中的指针 d_inode 指向 inode 结构，这是一对一的关系，因为一个目录项只代表着一个文件。可是，反过来就不一样了，同一个文件可以有多个不同的文件名或路径（通过系统调用 link()建立，可是注意与“符号连接”的区别，那是由 symlink()建立的），所以从 inode 结构到 dentry 结构的方向可以是一对多的关系。因此，inode 结构中的 i_dentry 是个队列，dentry 结构通过其队列头部 d_alias 挂入相应 inode 结构的队列中。

至于 d_rehash()则将 dentry 结构挂入杂凑队列，代码也在同一文件中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/dcache.c 847 860 ====================

[path_walk()>real_lookup()>ext2_lookup()>d_add()>d_rehash()]

```
/**
 * d_rehash   - add an entry back to the hash
 * @entry: dentry to add to the hash
 *
 * Adds a dentry to the hash according to its name.
 */

void d_rehash(struct dentry * entry)
{
     struct list_head *list = d_hash(entry->d_parent, entry->d_name.hash);
     spin_lock(&dcache_lock);
     list_add(&entry->d_hash, list);
     spin_unlock(&dcache_lock);
}
```

回到 real_lookup()的代码，现在已经找到了或者建立了所需的 dentry 结构，接着就返回到 path_walk()的代码中（见 fs/namei.c 中的 497 行）。

当前节点的 dentry 结构是有了，但是这个节点会不会是一个安装点呢？所以在 503 行调用 d_mountpoint()加以检验。

==================== include/linux/dcache.h 259 262 ====================

[path_walk()>d_mountpoint()]

```
static __inline__ int d_mountpoint(struct dentry *dentry)
{
     return !list_empty(&dentry->d_vfsmnt);
}
```

如果是安装点，就调用__follow_down()前进到所安装设备的根节点。这两个函数分别定义于 include/linux/dcache.h 和 fs/namei.c 中，读者可以参考“文件系统的安装与拆卸”一节。

最后，当前节点会不会只是代表着一个连接呢？对这种情况的检验取决于具体的文件系统。有些文件系统根本就不支持连接，那就已经最终找到了所需的节点，此时要调用 dput()递增 dentry 结构中的共享计数，因为此后 path_walk()不再使用这个数据结构了。如果具体的文件系统支持连接，那就通过 do_follow_link()处理（fs/namei.c）：

==================== fs/namei.c 312 325 ====================

[path_walk()>do_follow_link()]

```
static inline int do_follow_link(struct dentry *dentry, struct nameidata *nd)
{
     int err;
     if (current->link_count >= 8)
          goto loop;
     current->link_count++;
     UPDATE_ATIME(dentry->d_inode);
     err = dentry->d_inode->i_op->follow_link(dentry, nd);
     current->link_count--;
```

```
    return err;
loop:
    path_release(nd);
    return -ELOOP;
}
```

对连接链的长度要有个限制，否则就有可能陷入循环，这个上限是 8。具体对连接链的跟随由相应 inode_operations 结构中的函数指针 follow_link 所提供的函数完成。就 Ext2 文件系统来说，这个函数是 ext2_follow_link()，这是在 fs/ext2/symlink.c 中定义的：

==================== fs/ext2/symlink.c 35 38 ====================

```
struct inode_operations ext2_fast_symlink_inode_operations = {
    readlink: ext2_readlink,
    follow_link:  ext2_follow_link,
};
```

函数 ext2_follow_link()的代码也在同一文件中：

==================== fs/ext2/symlink.c 29 33 ====================

[path_walk()>do_follow_link()>ext2_follow_link()]

```
static int ext2_follow_link(struct dentry *dentry, struct nameidata *nd)
{
    char *s = (char *)dentry->d_inode->u.ext2_i.i_data;
    return vfs_follow_link(nd, s);
}
```

对于 Ext2 文件系统，连接目标的路径名在 ext2_inode_info 结构（inode 结构中的 union）的 i_data 字段中。代表着连接的节点并没有文件内容（数据），所以在索引节点中不需要存储有关各个存储区间的信息，而这些空间正好可以用来存储连接目标的路径名。这部分信息在前面的 ext2_read_inode()中作为 ext2_inode_info 结构的一部分被复制到 inode 结构里面的 union u 中。现在，就以此为目标调用 vfs_follow_link()来达到目的。

函数 vfs_follow_link()的代码在 fs/namei.c 中。值得注意的是，这里从 ext2_follow_link()中对 vfs_follow_link()的调用意味着从较低的层次上（具体的 Ext2 文件系统）回到了更高的 vfs 层。为什么呢？这是因为符号连接的目标有可能在另一个格式不同的文件系统中。可想而知，在 vfs_follow_link()中势必又要调用 path_walk()来找到代表着连接对象的 dentry 结构，事实也正是这样（fs/namei.c）：

==================== fs/namei.c 1942 1945 ====================

[path_walk()>do_follow_link()>ext2_follow_link()>vfs_follow_link()]

```
int vfs_follow_link(struct nameidata *nd, const char *link)
{
    return __vfs_follow_link(nd, link);
}
```

再往下看（fs/namei.c）：

==================== fs/namei.c 1906 1940 ====================

[path_walk()>do_follow_link()>ext2_follow_link()>vfs_follow_link()>__vfs_follow_link()]

```
static inline int
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
__vfs_follow_link(struct nameidata *nd, const char *link)
{
     int res = 0;
     char *name;
     if (IS_ERR(link))
          goto fail;

     if (*link == '/') {
          path_release(nd);
          if (!walk_init_root(link, nd))
               /* weird __emul_prefix() stuff did it */
               goto out;
     }
     res = path_walk(link, nd);
out:
     if (current->link_count || res || nd->last_type!=LAST_NORM)
          return res;
     /*
      * If it is an iterative symlinks resolution in open_namei() we
      * have to copy the last component. And all that crap because of
      * bloody create() on broken symlinks. Furrfu...
      */
     name = __getname();
     if (IS_ERR(name))
          goto fail_name;
     strcpy(name, nd->last.name);
     nd->last.name = name;
     return 0;
fail_name:
     link = name;
fail:
     path_release(nd);
     return PTR_ERR(link);
}
```

至此，对一个中间节点的搜索落实的过程就完成了。回到原先 path_walk()的代码中，那儿 533 行的 continue 语句使执行又回到 439 行的 for 循环开始处，继续处理路径名中的下一个节点。到最后一个节点时，就会转到标号为 last_component 或 last_with_slashes 处。我们继续在 path_walk()的代码中往下看（fs/namei.c）：

==================== fs/namei.c 536 611 ====================

[path_walk()]

```
536  last_with_slashes:
537            lookup_flags |= LOOKUP_FOLLOW | LOOKUP_DIRECTORY;
538  last_component:
539            if (lookup_flags & LOOKUP_PARENT)
540                 goto lookup_parent;
541            if (this.name[0] == '.') switch (this.len) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            default:
                break;
            case 2:
                if (this.name[1] != '.')
                    break;
                follow_dotdot(nd);
                inode = nd->dentry->d_inode;
                /* fallthrough */
            case 1:
                goto return_base;
        }
        if (nd->dentry->d_op && nd->dentry->d_op->d_hash) {
            err = nd->dentry->d_op->d_hash(nd->dentry, &this);
            if (err < 0)
                break;
        }
        dentry = cached_lookup(nd->dentry, &this, 0);
        if (!dentry) {
            dentry = real_lookup(nd->dentry, &this, 0);
            err = PTR_ERR(dentry);
            if (IS_ERR(dentry))
                break;
        }
        while (d_mountpoint(dentry) && __follow_down(&nd->mnt, &dentry))
            ;
        inode = dentry->d_inode;
        if ((lookup_flags & LOOKUP_FOLLOW)
           && inode && inode->i_op && inode->i_op->follow_link) {
            err = do_follow_link(dentry, nd);
            dput(dentry);
            if (err)
                goto return_err;
            inode = nd->dentry->d_inode;
        } else {
            dput(nd->dentry);
            nd->dentry = dentry;
        }
        err = -ENOENT;
        if (!inode)
            goto no_inode;
        if (lookup_flags & LOOKUP_DIRECTORY) {
            err = -ENOTDIR;
            if (!inode->i_op || !inode->i_op->lookup)
                break;
        }
        goto return_base;
no_inode:
        err = -ENOENT;
        if (lookup_flags & (LOOKUP_POSITIVE|LOOKUP_DIRECTORY))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
591                 break;
592             goto return_base;
593 lookup_parent:
594             nd->last = this;
595             nd->last_type = LAST_NORM;
596             if (this.name[0] != '.')
597                 goto return_base;
598             if (this.len == 1)
599                 nd->last_type = LAST_DOT;
600             else if (this.len == 2 && this.name[1] == '.')
601                 nd->last_type = LAST_DOTDOT;
602 return_base:
603             return 0;
604 out_dput:
605             dput(dentry);
606             break;
607         }
608         path_release(nd);
609 return_err:
610         return err;
611 }
```

路径名的末尾有个“/”字符，意味着路径的终点是个目录，并且，如果这个节点代表着一个连接就一定要前进到所连接的对象（也是个目录）。所以，在这种情况下把标志位 LOOKUP_FOLLOW 和 LOOKUP_DIRECTORY 都设成 1。

调用参数中的 LOOKUP_PARENT 标志位置 1 表示要寻找的并不是路径中的终点，而是它的上一层，所以转到 593 行的 lookup_parent 标号处，根据终点的节点名把 nameidata 结构中的 last_type 设置成 LAST_NORM、LAST_DOT 或者 LAST_DOTDOT。但是，nameidata 结构中的指针 dentry 此时仍指向上一层节点的 dentry 结构。

不过，一般情况下 LOOKUP_PARENT 标志位都是 0，要找的是路径名中的终点。将代码中的 541～581 行与 473～509 行作一比较，就可以发现这两部分代码几乎是一样的，所不同的只是：

(1) 对于中间节点调用 cached_lookup()和 real_lookup()时标志 LOOKUP_CONTINUE 为 1，而对终结节点调用这两个函数时这个标志位为 0。但是，这个标志位仅在所属文件系统（通过其 dentry_operations 结构中的函数指针 d_revalidate）提供目录项验证函数时才有用，Ext2 文件系统并不提供这个函数，并不对所找到的 dentry 结构加以验证，所以这个因素不起作用。

(2) 当中间节点代表着符号连接时，对 do_follow_link()的调用是无条件的（只要文件系统的 inode_operations 结构中提供了相应的函数）。相比之下，当终结节点代表着符号连接时，则仅当 LOOKUP_FOLLOW 标志位为 1 时才调用这个函数。

(3) 如果一个中间节点的 dentry 结构尚未与一个 inode 结构挂上钩，则搜索就无法继续下去了，所以 path_walk()立即就要出错返回，出错代码为-ENOENT。可是，对于终结节点就不同了，在有些情况下允许代表常规文件（而不是目录）终结节点的 dentry 结构没有与之挂钩的 inode 结构。我们称这样的 dentry 结构为“negative”，反之则为“positive”。有一个标志位称为 LOOKUP_POSITIVE，就是表示所欲寻找的节点必须具有 inode 结构。所以，对于终结节点，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

当不存在 inode 结构时就转向 584 行的标号 no_inode，在那里根据 LOOKUP_POSITIVE 和 LOOKUP_DIRECTORY 两个标志位来决定是出错返回或者正常返回。

(4) 最后，如果节点是一个目录，那就要依靠文件系统通过其 inode_operations 结构中的指针 lookup 提供的函数来读入这一层目录，并在这个目录中搜索。要是这个函数指针为 NULL，那么搜索也就不能延续了。对于中间节点这意味着搜索失败（531 行），而对于终结节点则只有在所要求的是个目录（LOOKUP_DIRECTORY 标志为 1）时才意味着失败（见 582～587 行）。

从 path_walk()返回时，函数值为 0 表示搜索成功，此时 nameidata 结构中的指针 dentry 指向目标节点（不一定是终结节点）的 dentry 结构，指针 mnt 指向目标节点所在设备的安装结构。同时，这个结构中的 last_type 表示最后一个节点的类型，节点名则在 qstr 结构 last 中。如果失败的话，则函数值为一负的出错代码，而 nameidata 结构中提供失败的节点名等信息。

根据给定路径名找到目标节点的 dentry 结构（以及 inode 结构）的过程，涉及与文件系统有关的几乎所有数据结构以及这些数据结构间的联系，搞懂了这个过程就对文件系统有了基本的理解。同时，path_init()和 path_walk()（以及将这二者包装在一起的 user_walk()）又是在各种与文件系统有关的系统调用中最广泛使用的函数。这里，我们略举数例，这些代码大都在 open.c 中：

==================== fs/open.c 340 368 ====================

```
asmlinkage long sys_chdir(const char * filename)
{
	int error;
	struct nameidata nd;
	char *name;

	name = getname(filename);
	error = PTR_ERR(name);
	if (IS_ERR(name))
		goto out;

	error = 0;
	if (path_init(name,LOOKUP_POSITIVE|LOOKUP_FOLLOW|LOOKUP_DIRECTORY,&nd))
		error = path_walk(name, &nd);
	putname(name);
	if (error)
		goto out;

	error = permission(nd.dentry->d_inode,MAY_EXEC);
	if (error)
		goto dput_and_out;

	set_fs_pwd(current->fs, nd.mnt, nd.dentry);

dput_and_out:
	path_release(&nd);
out:
	return error;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这是系统调用 chdir()的代码。这里的 permission()检查访问权限，读者可参阅下一节“访问权限与文件安全性”。函数 set_fs_pwd()将当前进程的 fs_struct 结构中的指针 pwd 和 pwdmnt 分别设置成由 nameidata 中提供的 dentry 指针和 vfsmount 指针。下面内容请读者自己看。

==================== fs/open.c 368 498 ====================

```
asmlinkage long sys_chmod(const char * filename, mode_t mode)
{
	struct nameidata nd;
	struct inode * inode;
	int error;
	struct iattr newattrs;

	error = user_path_walk(filename, &nd);
	if (error)
		goto out;
	inode = nd.dentry->d_inode;

	error = -EROFS;
	if (IS_RDONLY(inode))
		goto dput_and_out;

	error = -EPERM;
	if (IS_IMMUTABLE(inode) || IS_APPEND(inode))
		goto dput_and_out;

	if (mode == (mode_t) -1)
		mode = inode->i_mode;
	newattrs.ia_mode = (mode & S_IALLUGO) | (inode->i_mode & ~S_IALLUGO);
	newattrs.ia_valid = ATTR_MODE | ATTR_CTIME;
	error = notify_change(nd.dentry, &newattrs);

dput_and_out:
	path_release(&nd);
out:
	return error;
}
```

函数 notify_change()的作用见本章后面的另外几节。

==================== fs/open.c 560 571 ====================

```
asmlinkage long sys_chown(const char * filename, uid_t user, gid_t group)
{
	struct nameidata nd;
	int error;

	error = user_path_walk(filename, &nd);
	if (!error) {
		error = chown_common(nd.dentry, user, group);
		path_release(&nd);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    }
    return error;
}
```

==================== fs/open.c 38 52 ====================

```
asmlinkage long sys_statfs(const char * path, struct statfs * buf)
{
    struct nameidata nd;
    int error;

    error = user_path_walk(path, &nd);
    if (!error) {
        struct statfs tmp;
        error = vfs_statfs(nd.dentry->d_inode->i_sb, &tmp);
        if (!error && copy_to_user(buf, &tmp, sizeof(struct statfs)))
            error = -EFAULT;
        path_release(&nd);
    }
    return error;
}
```

5.3 访问权限与文件安全性

Unix 操作系统从一开始就在其文件系统中引入了“文件主”、“访问权限”等概念，并在此基础上实现了有利于提高文件安全性的机制。从那以后这些概念和机制就一直被继承下来并进一步得到改进和完善。即使在经过了二十多年以后的今天，而且在计算机系统的安全性已经成为一个突出问题的情况下，这一套机制仍然不失其先进性。尽管还存在一些缺点和需要进一步改进的地方，从总体上说还是瑕不掩瑜。与当今正在广泛使用的其它操作系统相比，可以说 Unix 的安全性总的来说至少不会差于这些系统；如果考虑到近年来在 Unix（以及 Linux，下同）中已经作出的改进以及不难作出的进一步改进，可以说 Unix 在安全性方面与任何其它系统相比都不逊色。同时，从当前流行的其它操作系统中，人们不难看出它们受 Unix 影响的或明或暗的痕迹。

Unix 文件系统的访问权限是一种二维结构。就同一个用户来说，对一个文件的访问分成读、写和执行三种方式，因而形成三种不同的权限；而就同一种访问方式来说，则又可因访问者的身份属于文件主、文件主的同组人以及其他用户（称为 other）而分别决定允许与否。这样一共就有 9 种组合，可以用 9 个二进制位来表示。早期的 Unix 是在 16 位结构的 PDP-11 机器上开发出来的，所以从那时起就一直用一个 16 位短整数来表示一个文件的访问“模式”，而将其中的低 9 位用于访问权限。当时比较流行的是八进制表示法，所以正好将这 9 位表示成三个八进制数位，从高到低分别用于文件主（u）、文件主的同组人（g）以及其他（o），而每个八进制数位中的三个二进制位则从高到低分别用于读、写和执行三种权限。这种表示方法一直沿用至今，例如在命令“chomd 644 file1”中的 644 就是这样三个 8 进制位。此外，这种把访问者区分为文件主、同组人以及其他用户，根据访问者的身份而分别决定其访问权限的方案称为“Discretionary Access Control 自主访问控制”，简写为 DAC。

这个方案的实施分成几个方面。首先，除用户名外，每个用户还被授予一个（在系统范围内）惟

一的用户号 uid，并且总是属于某一个用户组，而每一个用户组则有惟一的组号 gid，这些信息记录在相当于一个小数据库的文件/etc/passwd 中。其次，每个文件的索引节点中记录着文件主的 uid、gid 以及文件访问“模式”。还有，在每个进程的 task_stuct 结构中相应地设置了 uid 和 gid 等字段。每当用户通过登录进入系统并创立第一个 shell 进程时，就从/etc/passwd 中根据用户名查得其 uid 和 gid，并将其设置到该 shell 进程的 task_struct 结构中，从此以后便代代相传。最后，也是最重要的是，内核在执行用户进程访问文件的请求时要对比进程（和用户）的 uid、gid 以及文件的访问模式，以决定该进程是否对此文件具有所要求的访问权限。（实际上，进程的 task_struct 结构中还有 euid、egid、suid、sgid、fsuid 以及 fsgid 等字段，下面还要解释）。此外，uid 为 0 的用户为“超级用户”，而超级用户对任何文件都具有与文件主相同的权限。还要注意，用户名与用户号并不是一对一的关系，多个用户，甚至所有用户，都可以对应到同一个用户号。

由于超级用户的进程对任何文件都具有与文件主相同的权限，实际上可以对任何文件为所欲为，这就带来了危险（这里还没有考虑有人非法取得特权用户权限所引起的问题）。所以，有时候需要通过一个进程的用户号和组号来改变（限制）其访问极限。由此引申出了进程的“真实”用户号、“真实 ”组号和当前的“有效”用户号、“有效”组号的概念。相应地，在进程的 task_stuct 结构中也增设了 euid（表示“ effective uid”）和 egid 两个字段，并且提供了 setuid()、seteuid()等系统调用。另一方面，在改变“有效”用户号时往往需要把原来的“有效”用户号暂时保存起来，以便以后恢复，所以在 task_struct 结构中又增设了 suid（表示“ saved uid”）和 sgid 两个字段，这样，在 task_struct 结构中就有了三个用户号和三个组号，即 uid、euid、suid 以及 gid、egid、sgid。后来，在开发和使用网络文件系统 NFS 的“守护神”（即服务进程）的过程中又认识到，在网络环境下对文件的访问还需要一个不同的用户号，因此又增加一个 fsuid 和一个 fsgid。通常 fsuid 与 euid 相同，而 fsgid 与 egid 相同，但是在特殊的情况下可以不同。这里要指出，一般而言，只有特权用户以及具有特权用户权限的进程（见下面的所谓“ set_uid”文件和进程）才能通过系统调用来改变其用户号和组号，这些系统调用的结果都是使进程的权限更受限制；在相反的方向上，则最多是恢复到原有的水平，所以。一个非特权用户进程是不能通过 setuid()或 seteuid()得到特权用户的权限的，这一点跟读者头脑中一个普通用户可以通过 shell 命令“ su”变成特权用户的印象可能不一致。这里面的原因是 su 是一个“ set_uid”可执行程序，它的文件主是 root，即特权用户，所以当普通用户执行 su 的过程中就自动具有了特权用户的权限，这正是我们接下去要讨论的。

在前述二维访问权限机制的框架中，让我们考虑一个问题，即一个普通用户怎样才可以改变它自己的口令。我们知道，有关用户的名称、用户号、组号、口令等信息都保存在文件/etc/passwd 中。这个文件的主人只能是超级用户，因为只有超级用户才是系统中最核心、权力最大的用户，通常就是系统的管理员。除超级用户外，其它所有的用户对这个文件都不应该有“写”权限，因为那样的话每个用户都可以通过修改这个文件、将自己的用户号改成 0 而变成特权用户了。所以，除文件主以外，所有其他的用户对/etc/passwd 都只能有“读”权限而不能有“写”权限。这显然是合理的，而且只能如此。可是，这样一来，一个普通用户就不能通过运行一个什么程序来改变自己的口令了，因为改口令意味着改变/etc/passwd 中的内容。怎样解决这个矛盾呢？早期的 Unix 采用了一种在当时看来很巧妙的办法，就是在一些特殊用途的可执行文件上加一个标记，使得任何用户在执行这个文件（程序）时就暂时有了与该文件的文件主（通常是超级用户）相同的权限。这样，只要把用来改变口令的程序（/bin/passwd）加上这种标记，普通用户在执行这个程序的时候就能“拉大旗作虎皮”，暂时有了特权

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

用户的权限，可以改变/etc/passwd 的内容了。一旦执行完毕，则又回到原来的权限，又是普通用户了。这样的可执行文件，就称为“set_uid”文件，而加在这种文件上的标记，则是在文件模式中的一个标志位 S_ISUID。与此类似，还有一个标志位 SISGID，可以理解为对 S_ISUID 标志位的推广。有时候人们称 S_ISUID 标志位为“s”位，因为在用命令“ls -l”列目录时把表示这种文件对文件主的可执行权的字符“x”变成了“s”。在当时，这个办法确实很巧妙、很有效，据说 AT&T 还为此申请了专利。可是，近年来却发现这种“set_uid”文件给黑客们带来了可乘之机，简直已经成了 Unix（以及 Linux）在安全性方面的万恶之源，后面我们还会回到这个问题上来。

除这两个标志位以外，早期 Unix 还为可执行文件定义了一个“粘滞”（sticky）标志位。对于一些频繁运行的程序，可以把这个标志位设成 1，使得内核在这个程序运行完毕后尽可能将其映象保存在内存中不予释放，这样下一次需要启动这个程序运行时就不需要再从磁盘装入了。不过，现在的 Unix 和 Linux 都已采用虚存管理，所以这个标志位现在已经没有什么意义了。

前面讲过，文件的模式是以一个 16 位无符号整数表示的，其中 9 位已经用于对三种不同用户的访问权限，现在又用去了 3 位，这样还剩下 4 位，用来表示文件的类型。不过，由于只剩下 4 位，要为每种文件类型都分配一个标志位就不够了，所以表示文件类型的这 4 位是编码的。对文件类型和上述几个标志位的定义在 include/linux/stat.h 中，但是另一个文件 include/linux/sysv_fs.h 中有几行注释提供了比较详细的说明：

==================== include/linux/sysv_fs.h 255 266 ====================

```
/* The admissible values for i_mode are listed in <linux/stat.h> :
 * #define S_IFMT  00170000  mask for type
 * #define S_IFREG  0100000  type = regular file
 * #define S_IFBLK  0060000  type = block device
 * #define S_IFDIR  0040000  type = directory
 * #define S_IFCHR  0020000  type = character device
 * #define S_IFIFO  0010000  type = named pipe
 * #define S_ISUID  0004000  set user id
 * #define S_ISGID  0002000  set group id
 * #define S_ISVTX  0001000  save swapped text even after use
 * Additionally for SystemV:
 * #define S_IFLNK  0120000  type = symbolic link
```

注意，这里的数字均为八进制，其中 S_IFMT 并不代表一种文件类型，而只是对文件类型的屏蔽位段。对低 9 位的定义则为：

==================== include/linux/stat.h 32 45 ====================

```
#define S_IRWXU 00700
#define S_IRUSR 00400
#define S_IWUSR 00200
#define S_IXUSR 00100

#define S_IRWXG 00070
#define S_IRGRP 00040
#define S_IWGRP 00020
#define S_IXGRP 00010

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define S_IRWXO 00007
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001
```

这个 16 位的文件模式存储在每个文件的索引节点中，而每个进程则在其 task_struct 结构中有 uid、euid 等说明其身份的信息。这就是判定一个进程是否有权对某个文件进行某种访问的基础。对访问权限的判定主要是由函数 permission()完成的，读者在 path_walk()的代码中已经看到，在那里的 for 循环中对路径中的每一个节点调用这个函数，其代码存 fs/namei.c 中：

==================== fs/namei.c 183 193 ====================

```
int permission(struct inode * inode,int mask)
{
    if (inode->i_op && inode->i_op->permission) {
        int retval;
        lock_kernel();
        retval = inode->i_op->permission(inode, mask);
        unlock_kernel();
        return retval;
    }
    return vfs_permission(inode, mask);
}
```

参数 mask 为代表着所要求的访问方式的标志位，定义于 include/linux/fs.h 中：

==================== include/linux/fs.h 63 65 ====================

```
#define MAY_EXEC 1
#define MAY_WRITE 2
#define MAY_READ 4
```

对于一般的文件系统就分成这么三种方式。网络文件系统 NFS 的情况特殊，除这三种方式以外还定义了 MAY_TRUNC、MAY_LOCK 等方式以及这些方式的若干组合，不过 NFS 不在本书要讨论的范围内。

如果具体的文件系统通过其 inode_operations 结构中的函数指针 permisson 提供了特定的访问权限判定函数，那就把事情交给它了，否则就执行一般的 vfs_permission()。

就 Ext2 文件系统而言，共有三个 inode_operations 结构，即 ext2_file_inode_operations、ext2_dir_inode_operations 以及 ext2_fast_symlink_inode_operations，根据具体 inode 结构所代表的节点性质而在 ext2_read_inode()中将其 i_op 指针设置成指向这三者之一。可是，这三个结构中都没有提供专门的 permisson 操作（函数指针 permission 为 NULL），所以执行 vfs_permission()，其代码见 fs/namei.c。

==================== fs/namei.c 147 181 ====================

[permission()>vfs_permission()]

```
/*
 *  permission()
 *
 * is used to check for read/write/execute permissions on a file.
 * We use "fsuid" for this, letting us set arbitrary permissions
```

```c
 * for filesystem access without changing the "normal" uids which
 * are used for other things..
 */
int vfs_permission(struct inode * inode,int mask)
{
     int mode = inode->i_mode;

     if ((mask & S_IWOTH) && IS_RDONLY(inode) &&
           (S_ISREG(mode) || S_ISDIR(mode) || S_ISLNK(mode)))
          return -EROFS; /* Nobody gets write access to a read-only fs */

     if ((mask & S_IWOTH) && IS_IMMUTABLE(inode))
          return -EACCES; /* Nobody gets write access to an immutable file */

     if (current->fsuid == inode->i_uid)
          mode >>= 6;
     else if (in_group_p(inode->i_gid))
          mode >>= 3;

     if (((mode & mask & S_IRWXO) == mask) || capable(CAP_DAC_OVERRIDE))
          return 0;

     /* read and search access */
     if ((mask == S_IROTH) ||
         (S_ISDIR(inode->i_mode)  && !(mask & ~(S_IROTH | S_IXOTH))))
          if (capable(CAP_DAC_READ_SEARCH))
               return 0;

     return -EACCES;
}
```

这里用到的一些宏操作分别定义于 include/linux/fs.h 和 include/linux/stat.h：

==================== include/linux/fs.h 146 146 ====================

```c
146  #define IS_RDONLY(inode) ((inode)->i_sb->s_flags & MS_RDONLY)
```

==================== include/linux/stat.h 24 30 ====================

```c
24   #define S_ISLNK(m) (((m) & S_IFMT) == S_IFLNK)
25   #define S_ISREG(m) (((m) & S_IFMT) == S_IFREG)
26   #define S_ISDIR(m) (((m) & S_IFMT) == S_IFDIR)
27   #define S_ISCHR(m) (((m) & S_IFMT) == S_IFCHR)
28   #define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK)
29   #define S_ISFIFO(m)    (((m) & S_IFMT) == S_IFIFO)
30   #define S_ISSOCK(m)    (((m) & S_IFMT) == S_IFSOCK)
```

IS_RDONLY 表示节点所在的文件系统，即磁盘设备，是按“只读”方式安装的。在这样的磁盘上，对于常规文件、目录以及符号连接这三种节点都不能写。但是，即使是在按“只读”方式安装的文件系统中，如果节点所代表的是 FIFO 文件、插口等特殊文件，或者设备文件（块设备或字符设备都一样），那就未必是不可写的。为什么呢？因为对这些“文件”的写访问实际上不会或者不一定写到该节点所

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在的磁盘上去。

其次，IS_IMMUTABLE 的定义为：

==================== include/linux/fs.h 155 155 ====================

```
155 #define IS_IMMUTABLE(inode) ((inode)->i_flags & S_IMMUTABLE)
```

在较新的 Linux（和 Unix）版本中，除访问权限外又给每个文件加上了一些属性，“不可更改”即是其中之一。这些属性也像访问权限一样以标志位的形式存储在文件的索引节点中，但是不像访问权限那样区分文件主、文件主的同组用户以及公众，而是另成体系，并且凌驾于访问权限之上。而且，一旦设置了这些属性，即使是特权用户也不能在系统还在正常的多用户环境下运行时将这些属性去除。这样，就算有黑客偷到了特权用户的口令，对这些文件也就无能为力了。如果一个文件被设置成了“不可更改”，那么即使是超级用户通过 chmod()把文件的访问模式设置成“可写”也无济于事。显然，这是因安全性考虑而作的改进和增强。表示这些属性的标志位定义于 include/linux/ext2_fs.h：

==================== include/linux/ext2_fs.h 183 205 ====================

```
183 /*
184  * Inode flags
185  */
186 #define  EXT2_SECRM_FL          0x00000001 /* Secure deletion */
187 #define  EXT2_UNRM_FL           0x00000002 /* Undelete */
188 #define  EXT2_COMPR_FL          0x00000004 /* Compress file */
189 #define EXT2_SYNC_FL            0x00000008 /* Synchronous updates */
190 #define EXT2_IMMUTABLE_FL       0x00000010 /* Immutable file */
191 #define EXT2_APPEND_FL          0x00000020 /* writes to file may only append */
192 #define EXT2_NODUMP_FL          0x00000040 /* do not dump file */
193 #define EXT2_NOATIME_FL         0x00000080 /* do not update atime */
194 /* Reserved for compression usage... */
195 #define EXT2_DIRTY_FL           0x00000100
196 #define EXT2_COMPRBLK_FL        0x00000200 /* One or more compressed clusters */
197 #define EXT2_NOCOMP_FL          0x00000400 /* Don't compress */
198 #define EXT2_ECOMPR_FL          0x00000800 /* Compression error */
199 /* End compression flags --- maybe not all used */
200 #define EXT2_BTREE_FL           0x00001000 /* btree format dir */
201 #define EXT2_RESERVED_FL        0x80000000 /* reserved for ext2 lib */
202
203 #define EXT2_FL_USER_VISIBLE          0x00001FFF /* User visible flags */
204 #define EXT2_FL_USER_MODIFIABLE       0x000000FF /* User modifiable flags */
205
```

其中有些属性是为其它目的而设的（如压缩），我们在这里只关心与安全性有关的属性。例如，EXT2_APPEND_FL 表示对文件的写访问只能添加在文件的末尾，而不能改变文件中已有的内容。读者会问，在打开文件时不是就有个“添加”模式吗？为什么这里又要来一个“添加”属性呢？答案很简单，打开文件时的“添加”模式是用户进程“自愿”的，而文件的“添加”属性却是强制的。

所以，只要 inode 结构中的 i_flag 里面的 S_IMMUTABLE 标志为 1，那就剥夺了所有用户对这个文件的“写”访问权（见 163 行），而与文件所设置的访问权限以及访问者的身份无关。

还有个属性是 EXT2_NODUMP_FL，意图是使可执行文件在运行中访问内存出错（超界访问等）

PDF-XCHANGE Click to buy NOW! www.docu-track.com

时不要生成“dump”文件。从 Unix 的早期版本开始可执行文件在运行过程中因访问内存侵权而出错时都会把当时的内存映象卸载到一个磁盘文件中（名为“core”，因为早期的计算机采用磁芯存储器），使程序员可以使用调试工具（如 gdb 等）来重建起发生问题时的场景，这对于软件的维护显然是有好处的。可是，在实践中却发现，这也给怀有恶意的黑客们提供了可乘之机。为了在某些情况下不让产生 dump 文件，在 task_struct 结构中增设了一个标志位 dumpable，在某些情况下（例如通过 seteuid()设置了有效用户号）就将这个标志位清 0。同时，又设置了一个系统调用 prctl()，其用途之一就是将 dumpable 标志设置成 1 或 0，可是这还是不能解决防止恶意攻击的问题。在一些特殊的应用环境（如银行）中，，对一些特殊的可执行程序，需要完全杜绝其产生 dump 文件的可能性，这就是设置 EXT2_NODUMP_FL 属性及标志位的意图。此外，对于某些特殊文件，不能像对一般的文件那样每次访问后就要打下时间印记，标志位 EXT2_NOATIME_FL 就是为此目的而设置的。总之，对于传统的 Unix 文件系统而言，这些属性（标志位）都是“体制外”的，所以不能纳入原先的框架中，而其中有一些是为增强文件系统的安全性而设置的。

回到 permission()的代码中，下面就是访问权限的比对了。这里的 mode 是取自 inode 结构中的文件访问模式，即前述的 16 位无符号短整数；mask 则为所要求的访问方式，即 MAY_EXEC、MAY_WRITE 或 MAY_READ，实际上只用了最低 3 位。前面说过，当前进程的 fsuid 是专用于文件访问目的的有效 uid，通常与进程的 euid 相同，但是在使用网络文件系统时可能会不同。如果当前进程的 fsuid 与文件主的 uid 相同，那么要比对的是 mode 中用于文件主的访问权限，所以把 mode 右移 6 位，把用于文件主的三个标志位移到最低的 3 位中。如果当前进程的 fsuid 与文件主的 uid 不同，那就要检查一下当前进程所属用户的组号与文件主的组号是否相符，若二者相符，则适用同组人的访问权限，所以要把 mode 右移 3 位。判定是否同组比判定是否文件主要复杂一些，是通过一个函数 in_group_p()来完成的，其代码在 kernel/sys.c 中：

==================== kernel/sys.c 939 948 ====================

[permission()>vfs_permission()>in_group_p()]

```
/*
 * Check whether we're fsgid/egid or in the supplemental group..
 */
int in_group_p(gid_t grp)
{
     int retval = 1;
     if (grp != current->fsgid)
          retval = supplemental_group_member(grp);
     return retval;
}
```

如果进程的 fsgid 与文件主的组号相同，那就成了。可是即使这二者不同也还有可能实际上是相符的，因为一个用户（从而一个进程）可以同时属于若干个组，读者不妨回到“进程”一章看一下 sched.h 中对 task_struct 的定义，在 task_struct 结构中有个数组 groups[]，其大小为常数 NGROUPS，该常数在 include/asm-i386/param.h 定义为 32。当然，一个用户（进程）未必会那么“社会化”，所以在 task_struct 中还有个计数器 ngroups。与此相应，还提供了系统调用 get_groups()和 set_groups()。（只有得到授权的进程才可以 set_groups()）。所以，如果 fsgid 与文件主的组号不同，就要进一步拿这个数组中的其它“候

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

补”组号跟文件主的组号相比，函数 supplemental_group_member()的代码也在 sys.c 中：

==================== kernel/sys.c 923 937 ====================
[permission()>vfs_permission()>in_group_p()>supplemental_group_member()]

```
static int supplemental_group_member(gid_t grp)
{
	int i = current->ngroups;

	if (i) {
		gid_t *groups = current->groups;
		do {
			if (*groups == grp)
				return 1;
			groups++;
			i--;
		} while (i);
	}
	return 0;
}
```

最后，如果当前进程确实不属于文件主的同组人，那就是属于“其它”用户了。此时 mode 不需要移位，因为要比对的 3 位已经在最低的位置上了。

常数 S_IRWXO 的值为 7，所以比对的是此时 mode 中最低的 3 位。比对的结果相符时，permission()返回 0；要是不符呢？一般而言就失败了。但是还有例外。首先，如果当前进程得到了授权，允许其 CAP_DAC_OVERRIDE，即可以凌驾于文件系统的访问权限控制机制 DAC 之上，则基本上不受其限制。不过，前面 159 行和 163 行中检查的两种情况不在内。实际上，IS_IMMUTABLE 要有另一种授权（CAP_LINUX_IMMUTABLE）的进程才能设置。所以，这种进程就好像是捧着“尚方宝剑”的钦差大臣，这才是真正意义上的“超级用户”。可惜“超级用户”和“特权用户”这两个词都已经用于 uid 为 0 的用户，所以我们在书本中称此类进程为“授权进程”。等一下我们还要回到这个话题上来。

除了拥有 CAP_DAC_OVERRIDE 授权的进程以外，还有一种特殊情况，那就是另一种授权 CAP_DAC_READ_SEARCH，拥有这种特权的进程可以读任何文件，并且可以搜索任何目录节点，所以，代码中的 177 行检查所要求的是否读访问或者对目录节点的搜索。这里要提醒读者，搜索目录节点时所要求的访问方式为“执行”而不是“读”，所以在前一节里 path_walk()的 for 循环中对每个目录节点调用 permission()时的参数为 MAY_EXEC，而不是 MAY_READ。

如前所述，用户进程在一定条件下可以通过系统调用来设置其用户号，有关的系统调用有 setuid()、setfsuid()、seteuid()以及 setreuid()。其中 setuid()是标准的“设置用户号”调用，内核中与之相应的函数为 sys_setuid()，其代码在 sys.c 中：

==================== kernel/sys.c 548 583 ====================

```
/*
 * setuid() is implemented like SysV with SAVED_IDS
 *
 * Note that SAVED_ID's is deficient in that a setuid root program
 * like sendmail, for example, cannot set its uid to be a normal
 * user and then switch back, because if you're root, setuid() sets
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
554  * the saved uid too.  If you don't like this, blame the bright people
555  * in the POSIX committee and/or USG.  Note that the BSD-style setreuid()
556  * will allow a root program to temporarily drop privileges and be able to
557  * regain them by swapping the real and effective uid.
558  */
559 asmlinkage long sys_setuid(uid_t uid)
560 {
561      int old_euid = current->euid;
562      int old_ruid, old_suid, new_ruid;
563
564      old_ruid = new_ruid = current->uid;
565      old_suid = current->suid;
566      if (capable(CAP_SETUID)) {
567           if (uid != old_ruid && set_user(uid) < 0)
568                return -EAGAIN;
569           current->suid = uid;
570      } else if ((uid != current->uid) && (uid != current->suid))
571           return -EPERM;
572
573      current->fsuid = current->euid = uid;
574
575      if (old_euid != uid)
576           current->dumpable = 0;
577
578      if (!issecure(SECURE_NO_SETUID_FIXUP)) {
579           cap_emulate_setxuid(old_ruid, old_euid, old_suid);
580      }
581
582      return 0;
583 }
```

一般超级用户(其 euid 为 0)都具有 CAP_SETUID 授权，此时在 569 行和 573 行把当前进程的 euid、suid、fsuid 都设置成新的 uid。注意，这里把 suid 也设置成新的 uid，实在是个败笔，因为 task struct 结构中的 suid 本意在于“ save uid”，即在暂时改变 euid 时可以记住原来的 euid 是什么，以便以后恢复。而这里把 suid 也设置成了新的“ uid”，就失去了它的作用，并且用户号改变的历史也被一笔勾销，以后无法恢复成超级用户了。代码的原作者在函数前面加了注，也谈到了这个问题。可是，超级用户在调用 setuid()时把其 suid 也设置成新的 uid，这是在 POSIX 标准中规定了的，明知不合理也只能如此。正因为这样，在 BSD（以及 Linux）中另外提供了系统调用 seteuid()和 setreuid()来避免这个缺点。相比之下，对于不具备 CAP_SETUID 授权的进程，则只设置当前进程的 euid 和 fsuid，但是只有在新的 uid 就是进程的真实 uid 或者 suid 时才能进行。

每当进程改变其 euid 时，其 task_struct 结构中的标志位 dumpable 就被清 0，这样进程在访问出错时就不会产生 dump 文件了。

如果当前进程具有 CAP_SETUID 授权，并且新的 uid 又与原来的真实用户号不同，则连进程的真实用户号也要改变，这是通过 set_user()实施的（kernel/sys.c）：

==================== kernel/sys.c 466 486 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_setuid()>set user()]

```
static int set_user(uid_t new_ruid)
{
    struct user_struct *new_user, *old_user;

    /* What if a process setreuid()'s and this brings the
     * new uid over his NPROC rlimit?  We can check this now
     * cheaply with the new uid cache, so if it matters
     * we should be checking for it.  -DaveM
     */
    new_user = alloc_uid(new_ruid);
    if (!new_user)
        return -EAGAIN;
    old_user = current->user;
    atomic_dec(&old_user->processes);
    atomic_inc(&new_user->processes);

    current->uid = new_ruid;
    current->user = new_user;
    free_uid(old_user);
    return 0;
}
```

我们在“进程”一章中讲到过，内核中有个杂凑表 uidhash，各个进程的 task_struct 结构按其用户号 uid 的杂凑值挂入该杂凑表的某个队列中。这样，根据给定的 uid 就可以很快找到所有属于该用户的进程。同时，在 task_struct 结构中有个指针 user，指向一个 user_struct 数据结构，这个数据结构就好像 task_struct 结构与杂凑队列之间的连接件，进程的 task_struct 结构就是通过它挂入杂凑队列。现在，既然当前进程要“改换门庭”了，就要从原来的杂凑队列中脱链并将其 user_sturct 结构释放，然后另行分配一个 user_strct 结构并挂入另一个队列。函数 free_uid()的代码也在 kernel/fork.c 中：

==================== kernel/user.c 76 83 ====================
[sys_setuid()>set_user()>free_uid()]

```
void free_uid(struct user_struct *up)
{
    if (up && atomic_dec_and_lock(&up->__count, &uidhash_lock)) {
        uid_hash_remove(up);
        kmem_cache_free(uid_cachep, up);
        spin_unlock(&uidhash_lock);
    }
}
```

函数 alloc_uid()的作用与 free_uid()正好相反，我们就不列出它的代码了。

我们在前面讲过，超级用户的进程通常是得到某些授权的。相比之下，一般用户则得不到任何授权，它所有的只是文件系统的访问权限机制 DAC 所赋予的基本权利，而且这些基本权利也有可能得不到兑现，因为文件的属性如 IS_IMMUTABLE 等是凌驾于 DAC 之上的。可想而知，当一个超级用户进程改变其 uid 至某一普通用户时，其授权也要发生一些变化。

对进程的授权是独立于文件系统的访问权限控制以外，并且凌驾于其上的机制。为此目的在 task_strucl 结构中设置了 cap_effctive、cap_inheritable 和 cap_permitted 三个字段，其类型为 kernel_capt_t，目前实际上是 32 位无符号整数。每一种授权（capability）都用一个标志位来表示，目前共定义了 29 种授权，所以 32 位无符号整数就够用了。这些标志位（和授权）的定义在 include/linux/capability.h 中：

==================== include/linux/capability.h 65 278 ====================

```
/**
 ** POSIX-draft defined capabilities.
 **/

/* In a system with the [_POSIX_CHOWN_RESTRICTED] option defined, this
   overrides the restriction of changing file ownership and group
   ownership. */

#define CAP_CHOWN            0

/* Override all DAC access, including ACL execute access if
   [_POSIX_ACL] is defined. Excluding DAC access covered by
   CAP_LINUX_IMMUTABLE. */

#define CAP_DAC_OVERRIDE     1

/* Overrides all DAC restrictions regarding read and search on files
   and directories, including ACL restrictions if [_POSIX_ACL] is
   defined. Excluding DAC access covered by CAP_LINUX_IMMUTABLE. */

#define CAP_DAC_READ_SEARCH  2

/* Overrides all restrictions about allowed operations on files, where
   file owner ID must be equal to the user ID, except where CAP_FSETID
   is applicable. It doesn't override MAC and DAC restrictions. */

#define CAP_FOWNER           3

/* Overrides the following restrictions that the effective user ID
   shall match the file owner ID when setting the S_ISUID and S_ISGID
   bits on that file; that the effective group ID (or one of the
   supplementary group IDs) shall match the file owner ID when setting
   the S_ISGID bit on that file; that the S_ISUID and S_ISGID bits are
   cleared on successful return from chown(2) (not implemented). */

#define CAP_FSETID           4

/* Used to decide between falling back on the old suser() or fsuser(). */

#define CAP_FS_MASK          0x1f

/* Overrides the restriction that the real or effective user ID of a
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
   process sending a signal must match the real or effective user ID
   of the process receiving the signal. */

#define CAP_KILL           5

/* Allows setgid(2) manipulation */
/* Allows setgroups(2) */
/* Allows forged gids on socket credentials passing. */

#define CAP_SETGID         6

/* Allows set*uid(2) manipulation (including fsuid). */
/* Allows forged pids on socket credentials passing. */

#define CAP_SETUID         7


/**
 ** Linux-specific capabilities
 **/

/* Transfer any capability in your permitted set to any pid,
   remove any capability in your permitted set from any pid */

#define CAP_SETPCAP         8

/* Allow modification of S_IMMUTABLE and S_APPEND file attributes */

#define CAP_LINUX_IMMUTABLE  9

/* Allows binding to TCP/UDP sockets below 1024 */
/* Allows binding to ATM VCIs below 32 */

#define CAP_NET_BIND_SERVICE 10

/* Allow broadcasting, listen to multicast */

#define CAP_NET_BROADCAST    11

/* Allow interface configuration */
/* Allow administration of IP firewall, masquerading and accounting */
/* Allow setting debug option on sockets */
/* Allow modification of routing tables */
/* Allow setting arbitrary process / process group ownership on
   sockets */
/* Allow binding to any address for transparent proxying */
/* Allow setting TOS (type of service) */
/* Allow setting promiscuous mode */
/* Allow clearing driver statistics */
```

```
/* Allow multicasting */
/* Allow read/write of device-specific registers */
/* Allow activation of ATM control sockets */

#define CAP_NET_ADMIN        12

/* Allow use of RAW sockets */
/* Allow use of PACKET sockets */

#define CAP_NET_RAW          13

/* Allow locking of shared memory segments */
/* Allow mlock and mlockall (which doesn't really have anything to do
   with IPC) */

#define CAP_IPC_LOCK         14

/* Override IPC ownership checks */

#define CAP_IPC_OWNER        15

/* Insert and remove kernel modules - modify kernel without limit */
/* Modify cap_bset */
#define CAP_SYS_MODULE       16

/* Allow ioperm/iopl access */
/* Allow sending USB messages to any device via /proc/bus/usb */

#define CAP_SYS_RAWIO        17

/* Allow use of chroot() */

#define CAP_SYS_CHROOT       18

/* Allow ptrace() of any process */

#define CAP_SYS_PTRACE       19

/* Allow configuration of process accounting */

#define CAP_SYS_PACCT        20

/* Allow configuration of the secure attention key */
/* Allow administration of the random device */
/* Allow examination and configuration of disk quotas */
/* Allow configuring the kernel's syslog (printk behaviour) */
/* Allow setting the domainname */
/* Allow setting the hostname */
/* Allow calling bdflush() */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* Allow mount() and umount(), setting up new smb connection */
/* Allow some autofs root ioctls */
/* Allow nfsservctl */
/* Allow VM86_REQUEST_IRQ */
/* Allow to read/write pci config on alpha */
/* Allow irix_prctl on mips (setstacksize) */
/* Allow flushing all cache on m68k (sys_cacheflush) */
/* Allow removing semaphores */
/* Used instead of CAP_CHOWN to "chown" IPC message queues, semaphores
   and shared memory */
/* Allow locking/unlocking of shared memory segment */
/* Allow turning swap on/off */
/* Allow forged pids on socket credentials passing */
/* Allow setting readahead and flushing buffers on block devices */
/* Allow setting geometry in floppy driver */
/* Allow turning DMA on/off in xd driver */
/* Allow administration of md devices (mostly the above, but some
   extra ioctls) */
/* Allow tuning the ide driver */
/* Allow access to the nvram device */
/* Allow administration of apm_bios, serial and bttv (TV) device */
/* Allow manufacturer commands in isdn CAPI support driver */
/* Allow reading non-standardized portions of pci configuration space */
/* Allow DDI debug ioctl on sbpcd driver */
/* Allow setting up serial ports */
/* Allow sending raw qic-117 commands */
/* Allow enabling/disabling tagged queuing on SCSI controllers and sending
   arbitrary SCSI commands */
/* Allow setting encryption key on loopback filesystem */

#define CAP_SYS_ADMIN       21

/* Allow use of reboot() */

#define CAP_SYS_BOOT        22

/* Allow raising priority and setting priority on other (different
   UID) processes */
/* Allow use of FIFO and round-robin (realtime) scheduling on own
   processes and setting the scheduling algorithm used by another
   process. */

#define CAP_SYS_NICE        23

/* Override resource limits. Set resource limits. */
/* Override quota limits. */
/* Override reserved space on ext2 filesystem */
/* NOTE: ext2 honors fsuid when checking for resource overrides, so
   you can override using fsuid too */
```

```
/* Override size restrictions on IPC message queues */
/* Allow more than 64hz interrupts from the real-time clock */
/* Override max number of consoles on console allocation */
/* Override max number of keymaps */

#define CAP_SYS_RESOURCE     24

/* Allow manipulation of system clock */
/* Allow irix_stime on mips */
/* Allow setting the real-time clock */

#define CAP_SYS_TIME        25

/* Allow configuration of tty devices */
/* Allow vhangup() of tty */

#define CAP_SYS_TTY_CONFIG   26

/* Allow the privileged aspects of mknod() */

#define CAP_MKNOD           27

/* Allow taking of leases on files */

#define CAP_LEASE           28
```

代码的作者已经加了详尽的注释，我们这里就不作解释了。定义中的数值为标志位的位置，如 CAP_CHOWN 的定义为 0，即第 0 位。对授权的检查是由 capable()完成的，这是个 inline 函数，定义于 sched.h 中：

==================== include/linux/sched.h 681 699 ====================

```
/*
 * capable() checks for a particular capability.
 * New privilege checks should use this interface, rather than suser() or
 * fsuser(). See include/linux/capability.h for defined capabilities.
 */

static inline int capable(int cap)
{
#if 1 /* ok now */
     if (cap_raised(current->cap_effective, cap))
#else
     if (cap_is_fs_cap(cap) ? current->fsuid == 0 : current->euid == 0)
#endif
     {
          current->flags |= PF_SUPERPRIV;
          return 1;
     }
     return 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

这里的 cap_raised()是个宏操作，与之有关的定义都在 capability.h 中

==================== include/linux/capability.h 298 298 ====================

```
#define cap_t(x) (x)
```

==================== include/linux/capability.h 307 307 ====================

```
#define CAP_TO_MASK(x) (1 << (x))
```

==================== include/linux/capability.h 310 310 ====================

```
#define cap_raised(c, flag)  (cap_t(c) & CAP_TO_MASK(flag))
```

全局量 cap_bset 则设置成 CAP_FULL_SET，即全部标志位均为 1。

当进程改变其 uid 时，要通过 cap_emulate_setxuid()检查并可能改变其授权情况，除非在编泽内核前将一个常数 SECUREBITS_DEFAULT 中的 SECURE_NO_SETUID_FIXUP 标志位设置成为 1，表示可以忽略对进程的授权机制。函数 cap_emulate_setxuid()的代码在 sys.c 中：

==================== kernel/sys.c 420 464 ====================

[sys_setuid()>cap_emulate_setxuid()]

```
/*
 * cap_emulate_setxuid() fixes the effective / permitted capabilities of
 * a process after a call to setuid, setreuid, or setresuid.
 *
 *  1) When set*uiding _from_ one of {r,e,s}uid == 0 _to_ all of
 *  {r,e,s}uid != 0, the permitted and effective capabilities are
 *  cleared.
 *
 *  2) When set*uiding _from_ euid == 0 _to_ euid != 0, the effective
 *  capabilities of the process are cleared.
 *
 *  3) When set*uiding _from_ euid != 0 _to_ euid == 0, the effective
 *  capabilities are set to the permitted capabilities.
 *
 *  fsuid is handled elsewhere. fsuid == 0 and {r,e,s}uid!= 0 should
 *  never happen.
 *
 *  -astor
 *
 * cevans - New behaviour, Oct '99
 * A process may, via prctl(), elect to keep its capabilities when it
 * calls setuid() and switches away from uid==0. Both permitted and
 * effective sets will be retained.
 * Without this change, it was impossible for a daemon to drop only some
 * of its privilege. The call to setuid(!=0) would drop all privileges!
 * Keeping uid 0 is not an option because uid 0 owns too many vital
 * files..
 * Thanks to Olaf Kirch and Peter Benie for spotting this.
 */
extern inline void cap_emulate_setxuid(int old_ruid, int old_euid,
                                int old_suid)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
451  {
452      if ((old_ruid == 0 || old_euid == 0 || old_suid == 0) &&
453          (current->uid != 0 && current->euid != 0 && current->suid != 0) &&
454          !current->keep_capabilities) {
455           cap_clear(current->cap_permitted);
456           cap_clear(current->cap_effective);
457      }
458      if (old_euid == 0 && current->euid != 0) {
459           cap_clear(current->cap_effective);
460      }
461      if (old_euid != 0 && current->euid == 0) {
462           current->cap_effective = current->cap_permitted;
463      }
464  }
```

代码中的 cap_clear()是个宏操作，定义于 capability.h：

==================== include/linux/capability.h 343 343 ====================

```
343  #define cap_clear(c)       do { cap_t(c) =  0; } while(0)
```

结合代码作者所加的注释，读者应该可以读懂这段代码。举例来说，如果一个超级用户进程通过 seteuid()将进程的有效用户号 euid 从 0 改变成某个普通用户的用户号（非 0），则进程的“有效授权 ” cap_effective 变成 0，但是 cap_permitted 并未改变。以后，当这个进程恢复原先的 euid 时，就将进程的 cap_permitted 复制到 cap_effective 中。但是，setuid()将进程的 uid、euid 以及 suid 全部改变，所以进程的 cap_permitted 和 cap_effective 二者都被清 0，以后就不能恢复了。不过这里还有个例外，那就是进程的 task_struct 结构中有个标志位 keep_capabilities，可以通过系统调用 prctl()将这个标志位预先设成 1，这样就可以避免将 cap_permitted 清 0(注意第 458 行的 if 前面并没有 else，所以仍会将 cap_effective 清 0)。

读者可能会因为曾经使用 shell 命令 “ su” 升格成超级用户而得出一个错觉，似乎普通用户的进程也可以通过系统调用 setuid()将自己的用户号设置成 0 而变成超级用户进程。其实，/bin/su 是个属于超级用户的 “ set uid” 可执行程序，普通用户的进程在执行这个程序时就有了超级用户的 “身份”。在检查了口令以后，它就 fork()出一个新的 shell 进程。这个新 shell 进程的父进程具有超级用户的身份，所以它也成了超级用户进程。至于原来的 shell 进程和 su 进程，则都在睡眠等待(直至新的 shell 进程 exit())。从效果上看，就好像新的 shell 进程从原先的 shell 进程手中“接管”了终端的键盘和显示屏；而从用户界面来看，则似乎原来的 shell 进程“升格”成了超级用户进程。

在常规的访问权限控制机制 DAC 的基础上，有些 Unix 变种版本（如 AIX，Solaris 等）作了一个重要的改进，叫做“访问控制单”(Access Control List)，缩写为 ACL。在实现了 ACL 的系统中，每个文件可以伴随存储一份访问控制单，里面有一些“访问控制项”(Access Control Entry)，可以为具体的用户规定对基本访问权限的修正。例如，可以这样规定：当用户 A 属于用户组 g1 时就剥夺它的读写访问权；对于用户 B 则永远增加写访问权，而不论其是否为文件主或同组人。显然，对于商务应用这是很有意义的，可以改善文件系统的安全性。当前的 Linux 版本正在朝实现 ACL 迈进，在一些数据结构中已经设置了用于 ACL 的结构成分(如 ext2_inode 结构中的 i_file_acl 和 i_dir_acl)，以及 ext2_acl_entry 的数据结构，但是其代码则尚未实现，所以在函数 permission()中并未访问目标节点的 ACL。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

我们在前面看到了当进程改变用户号时授权的改变，可是这些授权最初是怎么来的呢？让我们来看当进程通过 exec()执行一个可执行文件时的情况，因为每一个进程初始的授权最终都可以追溯到这里，例如，当一个用户 login 进入系统时，系统就会 fork()出一个进程并让它执行/bin/bash（或 csh 等），而这个 shell 进程就成为该用户启动的所有进程的祖先。在“进程”一章中，读者已经看到在 do_execve()中要先通过 prepare_binprm()设置一个 linux_binprm 结构，在这个数据结构中同样有 cap_effective、cap_permitted 和 cap_inheritable 三字段，分别与 task_struct 结构中的三个字段相对应。函数 prepare_binprm()中与授权有关的处理为（见 fs/exec.c）：

==================== fs/exec.c 600 601 ====================

[sys_execve()>do_execve()>prepare_binprm()]

```
int prepare_binprm(struct linux_binprm *bprm)
{
```

==================== fs/exec.c 612 654 ====================

```
	bprm->e_uid = current->euid;
	bprm->e_gid = current->egid;

	if(!IS_NOSUID(inode)) {
		/* Set-uid? */
		if (mode & S_ISUID)
			bprm->e_uid = inode->i_uid;

		/* Set-gid? */
		/*
		 * If setgid is set but no group execute bit then this
		 * is a candidate for mandatory locking, not a setgid
		 * executable.
		 */
		if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP))
			bprm->e_gid = inode->i_gid;
	}

	/* We don't have VFS support for capabilities yet */
	cap_clear(bprm->cap_inheritable);
	cap_clear(bprm->cap_permitted);
	cap_clear(bprm->cap_effective);

	/*  To support inheritance of root-permissions and suid-root
	   *  executables under compatibility mode, we raise all three
	   *  capability sets for the file.
	   *
	   *  If only the real uid is 0, we only raise the inheritable
	   *  and permitted sets of the executable file.
	   */

	if (!issecure(SECURE_NOROOT)) {
		if (bprm->e_uid == 0 || current->uid == 0) {
			cap_set_full(bprm->cap_inheritable);
```

```
646                 cap_set_full(bprm->cap_permitted);
647             }
648             if (bprm->e_uid == 0)
649                 cap_set_full(bprm->cap_effective);
650         }
651
652         memset(bprm->buf,0,BINPRM_BUF_SIZE);
653         return kernel_read(bprm->file,0,bprm->buf,BINPRM_BUF_SIZE);
654     }
```

我们先从 631 行看起，等一下还要回到它前面的几行。三个字段全部清 0，这就是普通用户得到的授权。然后，如果进程在执行该可执行程序时的有效用户号为 0，则将 linux_binprm 结构中的 cap_inheritable 和 cap_effective 设成全 1；如果进程的真实用户号为 0 但有效用户号不为 0，则仅将 cap_inheritabte 设成全 1。也就是说，超级用户进程最初时具有全部授权。但是在开始运行过程后超级用户进程可以通过系统调用 capset()来减少其授权，改变以后进程的用户号仍旧是 0，还是超级用户进程，但是授权却减小了。注意，capset()只能减少而不能增加一个进程已有的授权，所以是单向的。

回到前面的第 615 行至 628 行，可以看到（可执行文件的）模式 mode 中的 S_ISGID 标志位怎样影响着 linux_binprm 结构中的用户号 e_uid 和组号 e_gid。

设置好 linux_binprm 结构，并且装入了可执行文件的映象以后，内核会通过一个函数 compute_creds()，根据 linux_binprm 结构中的内容来设置当前进程的 task_struct 结构中的相应内容，这个函数的代码在 exec.c 中：

==================== fs/exec.c 656 720 ====================

[load_aout_binary()>compute_creds()] 或 [load_elf_binary()>compute_creds()]

```
/*
 * This function is used to produce the new IDs and capabilities
 * from the old ones and the file's capabilities.
 *
 * The formula used for evolving capabilities is:
 *
 *       pI' = pI
 * (***) pP' = (fP & X) | (fI & pI)
 *       pE' = pP' & fE          [NB. fE is 0 or ~0]
 *
 * I=Inheritable, P=Permitted, E=Effective // p=process, f=file
 * ' indicates post-exec(), and X is the global 'cap_bset'.
 *
 */

void compute_creds(struct linux_binprm *bprm)
{
    kernel_cap_t new_permitted, working;
    int do_unlock = 0;

    new_permitted = cap_intersect(bprm->cap_permitted, cap_bset);
    working = cap_intersect(bprm->cap_inheritable,
```

```
                  current->cap_inheritable);
     new_permitted = cap_combine(new_permitted, working);

     if (bprm->e_uid != current->uid || bprm->e_gid != current->gid ||
         !cap_issubset(new_permitted, current->cap_permitted)) {
              current->dumpable = 0;

          lock_kernel();
          if (must_not_trace_exec(current)
              || atomic_read(&current->fs->count) > 1
              || atomic_read(&current->files->count) > 1
              || atomic_read(&current->sig->count) > 1) {
               if(!capable(CAP_SETUID)) {
                    bprm->e_uid = current->uid;
                    bprm->e_gid = current->gid;
               }
               if(!capable(CAP_SETPCAP)) {
                    new_permitted = cap_intersect(new_permitted,
                                   current->cap_permitted);
               }
          }
          do_unlock = 1;
     }


     /* For init, we want to retain the capabilities set
        * in the init_task struct. Thus we skip the usual
        * capability rules */
     if (current->pid != 1) {
          current->cap_permitted = new_permitted;
          current->cap_effective =
               cap_intersect(new_permitted, bprm->cap_effective);
     }

       /* AUD: Audit candidate if current->cap_effective is set */

       current->suid = current->euid = current->fsuid = bprm->e_uid;
       current->sgid = current->egid = current->fsgid = bprm->e_gid;

     if(do_unlock)
          unlock_kernel();
     current->keep_capabilities = 0;
}
```

前面在 prepare_binprm()设置了 linux_binprm 结构中的这些授权以后，还要与进程当前已有的授权进行一些整合。这里的 cap_intersect()和 cap_combine()都是 inline 函数，还有个 cap_issubset 则为宏操作，均定义于 capability.h：

==================== include/linux/capability.h 312 324 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static inline kernel_cap_t cap_combine(kernel_cap_t a, kernel_cap_t b)
{
     kernel_cap_t dest;
     cap_t(dest) = cap_t(a) | cap_t(b);
     return dest;
}

static inline kernel_cap_t cap_intersect(kernel_cap_t a, kernel_cap_t b)
{
     kernel_cap_t dest;
     cap_t(dest) = cap_t(a) & cap_t(b);
     return dest;
}
```

==================== include/linux/capability.h 341 341 ====================

```
#define cap_issubset(a,set)  (!(cap_t(a) & ~cap_t(set)))
```

也就是说，对于普通用户整合以后仍为 0，而对于超级用户则整合以后为当前的 cap_permitted 与当前的 cap_inheritable 的逻辑和。如果这个逻辑和不是当前 cap_permitted 的一个子集，则意味着执行给定可执行程序时的授权将比进程现有的授权有所提高，所以把变量 cap_raised 设成 1。当然，如果已经通过 capset()将当前进程的 cap_inheritable 设置成 0，则这种授权的回升就不可能发生了。授权的回升一般是允许的，但是在第 674 行所列的五种情况下则是有条件地允许。如果当前的 cap_permitted 中不包括 CAP_SETPCAP 就不允许了。这里 IS_NOSUID 是个宏定义，表示 inode 所在的文件系统在安装时在 super_block 结构中将 MS_NOSUID 标志位设成了 1，使该文件系统中所有可执行文件的“ set_uid ”标志位都作废了。

1 号进程，即 init()进程，是特殊的，它的授权是在宏定义 INIT_TASK 中固定了的。其 cap_effective、cap_inheritable 和 cap_permitted 分别定为 CAP_INIT_EFF_SET、CAP_INIT_INH_SET 以及 CAP_FULL_SET。其中 CAP_FULL_SET 为全部授权，而 CAP_INIT_EFF_SET 和 CAP_INIT_INH_SET 则为除 CAP_SETPCAP 以外的全部授权。系统中所有其它的进程都是 init()进程的后裔，所以只要 init()进程调用 capset()清除其 cap_effective 中的 CAP_SETPCAP，则以后 fork()出来的所有进程就都没有了这种授权。又如，只有具有 CAP_LINUX_IMMUTABLE 授权的进程才能改变文件的 S_IMMUTABLE 和 S_APPEND 属性，如果 init()进程将/etc/inetd.conf 的属性加上“不可改变”，把/var/log/messages 加上 S_APPEND 属性，然后清除其 CAP_LINUX_IMMUTABLE 授权，则以后 fork()出来的进程永远都不能改变这两个文件的这些属性了。显然，这是对于文件系统安全性的一大改进。

前面讲过，将可执行文件设置成“ set uid”模式，可以使执行它的进程在执行期间将其 euid 暂时改成该文件的文件主的 uid，实践中通常是使普通用户在执行某个可执行文件的期间变成超级用户。这在 Unix 的早期是一项很巧妙的发明，到现在也还有很重要的意义。但是，近年来的实践发现，对 Unix 系统的黑客攻击事件大多数都是与此有关的。这种攻击都与可执行程序本身的缺陷有关，其中最主要的就是所谓“缓冲区溢出”攻击。举例来说，可能会有这样一个应用程序：

```
main(int argc, char **agtv)
{
       char options[128];
       . . . . . .
```

```
        if (argv > 1) {
                strcpy(options, argv[1]);
        }
}
```

这段程序的开头将用户提供的命令行参数拷贝到一个大小为 128 字节的字符数组中，但是却没有检查字符串的长度，这是一个常见的错误。一般情况下，命令行参数不至于超过 128 字节，所以通常不会造成问题。可是，如果碰巧（或故意）命令行参数的字符串长度为 150 字节呢？这时候的 strcpy() 就越界了。由于字符数组 options 是在堆栈中的局部量，一越界以后就可能把 main()的返回地址也冲掉了。这一般会导致从 main()返回时访问出错，但是既然所要做的事情已经完成了，一般也就无所谓了。可是，黑客们有可能在经过多次试凑以后在堆栈中原先为 main()的返回地址的位置上有目的地植入一个返回地址（通过 strcpy()从命令行参数中复制进去），使得从 main()返回时就“返回”到一个特定的地方去。这个“特定的地方”通常也在堆栈中，并且通过类似的手段植入了一小段可执行代码，例如相当于编译后的“ system（"/bin/bash"）”这么二十来个字节。这么一来，当从 main()返回时就会 fork()出一个 shell 进程出来。对于一般的可执行文件，这么做的意义似乎并不大，因为既然你能启动这个可执行文件，就说明你本来就有个 shell 进程。可是，如果这个可执行文件是个“ set uid”文件呢？这时候黑客们从一个普通用户的 shell 进程开始，却以得到一个超级用户的 shell 进程而告终，因为这个 shell 进程是从超级用户退出之前 fork()的。当然，黑客们事先未必知道这个“ set uid”的可执行文件存在着这样的问题，他们可能是先通过一条 shell 命令“ find / -perm 00400 -uid 0 -type f -print”列出系统中所有属于超级用户的“ set uid”文件，然后逐一试凑而已，他们有的是时间！但是，一旦被他们发现这么一个可乘之机，那后果就严重了。得到了一个超级用户的 shell 进程以后，他们立即就会修改/etc/passwd，将他们的用户号改成 0，或者增加一个 uid 为 0 的新用户。从此以后，他们就可以“如入无人之境”了。堵塞这种漏洞的途径是多方面的，其中之一是把进程的堆栈段的属性改成“不可执行”，因为黑客们很难把什么内容植入到进程的代码段中。此外，准备用作“ set uid”可执行文件的应用程序要精心设计、精心实现和调试，并且系统中的“ set uid”可执行程序的数量要尽可能减少。

另一方面，可执行程序的“ set uid”机制是否真的必要也是个问题。我们在本节开头处以改变用户的口令为例来说明其必要性，但那是建立在本用户的进程要直接修改/etc/passwd 这么个前提之上的。早期 Unix 的进程间通信机制比较薄弱，所以别无他法。可是，现在的进程间通信机制已经很强，有的事情可以按 Client/Server 的模式来设计和实现。例如，现在完全可能在系统中建立一个内核线程 passwd_d 作为口令服务器，每当用户要改变口令时就通过插口与它建立起连接，然后由 passwd_d 负责来判定该用户是否可以改变其口令，如果可以就由它来修改/etc/passwd。这样，在用户进程与文件/etc/passwd 之间就隔了一层类似“防火墙”那样的东西，连对/etc/passwd 的“读”访问权也不必有了。

计算机系统的安全性是个综合性的问题，在相当程度上是个管理问题，而从技术角度来看则主要有两个方面的问题，即文件系统的安全性与网络操作的安全性。由于篇幅的限制，我们在本书中基本上不涉及有关网络的内容（我们计划另外写一本书专门介绍 Linux 内核中与网络有关的代码，其篇幅可能不小于本书。在那本书中我们将讨论 Linux 的网络操作安全性）。即使是文件安全性的问题，也小是在区区数十页的篇幅中能够讲清楚、讲全面的，所以有兴趣或有需要的读者可以参考有关的专著，例如 Simon Garfinkel 和 Gene Spafford 的 *Practical Unix & Internet Security*，就是一本被誉为这方面的“圣经”式的著作。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

有一种说法，说由于 Linux 内核的源代码公开而使其安全性降低了，因为黑客们可以从源代码中去寻找漏洞或可乘之机。这种说法理所当然地遭到持相反意见的人士的驳斥。公开的源代码固然使攻击的一方容易找到可乘之机，但是同时也使防守的一方易于事先防范。即使出了问题，事后的分析和解决问题也比较容易，毕竟防守方打的是“人民战争”。如果把攻防双方的较量比喻作一种“振荡”的话，则公开的源代码使振幅的“衰减”或“收敛”加快，这应该是好事。对于防守来说，攻、防双方都在明处总比都在暗中要好。事实上，最令人不安的就是“黑盒子”，两眼一抹黑不知道里面在干些什么。更何况，还有“官匪一家”，开发系统的人自己在系统中留下“后门”的可能。

以前，人们还只是从理论上谈论这种可能性，可是前不久报载有个 Microsoft 的工程师承认自己确曾在 Windows 中留下了“后门”。对于一个不公开源代码的系统，你怎么知道这样的“后门”到底有多少呢？

顺便多讲几句。前一阵（2000 年 10 月）又有关于 Microsoft 的报道，说发现有来自俄国的黑客侵入 Microsoft 的计算机系统达两个星期之久。对于所造成的损害则说法不一。同是 Microsoft 的官员，有的说黑客可能已经窃得 Windows 操作系统的源代码；有的则说黑客所偷取的只是正在开发中的某应用软件的源代码，而不是 Windows 的源代码，所以用户仍可放心云云。可是，不管这一次黑客是否已经得手（也许无人确切知道），这种可能性总不能讲没有。那样，终有一天，当用户还“放心”地守着“黑盒子”时，黑客们却手中有了 Windows 的源代码，并已作了分析研究。到那时，黑客们不出手便罢，一出手就招招都能点中“穴位”。如果他们想要闯进来的话，那可就真的可以如入无人之境了。

5.4 文件系统的安装和拆卸

在一个块设备（见本书下册“设备驱动”一章）上按一定的格式建立起文件系统的时候，或者系统引导之初，设备上的文件和节点都还是不可访问的。也就是说，还不能按一定的路径名访问其中特定的节点或文件（虽然作为“设备”是可访问的）。只有把它“安装”到计算机系统的文件系统中某个节点上，才能使设备上的文件和节点成为可访问的。经过安装以后，设备上的“文件系统”就成为整个文件系统的一部分，或者说一个子系统。一般而言，文件系统的结构就好像一棵倒立的树，不过由于可能存在着的节点间的“连接”和“符号连接”而并不一定是严格的图论意义上的“树”。最初时，整个系统中只有一个节点，那就是整个文件系统的“根”节点“/”，这个节点存在于内存中，而不在任何具体的设备上。系统在初始化时将一个“根设备”安装到节点“/”上，这个设备上的文件系统就成了整个系统中原始的、基本的文件系统（所以才称为根设备）。此后，就可以由超级用户进程通过系统调用 mount()把其它的子系统安装到已经存在于文件系统中的空闲节点上，使整个文件系统得以扩展，当不再需要使用某个子系统时，或者在关闭系统之前，则通过系统调用 umount()把已经安装的设备逐个“拆卸”下来。

系统调用 mount()将一个可访问的块设备安装到一个可访问的节点上。所谓“可访问”是指该节点或文件已经存在于已安装的文件系统中，可以通过路径名寻访。Unix（以及 Linux）将设备看作一种特殊的文件，并在文件系统中有代表着具体设备的节点，称为“设备文件”，通常都在目录“/dev”中。例如 IDE 硬盘上的第一个分区就是/dev/hda1。每个设备文件实际上只是一个索引节点，节点中提供了设备的“设备号”，由“主设备号”和“次设备号”两部分构成。其中主设备号指明了设备的种类，或

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

者更确切地说是指明了应该使用哪一组驱动程序。同一个物理的设备，如果有两组不同的驱动程序，在逻辑上就被视作两种不同的设备而在文件系统中有两个不同的“设备文件”。次设备号则指明该设备是同种设备中的第几个。所以，只要找到代表着某个设备的索引节点，就知道该怎样读 / 写这个设备了。既然是一个“可访问”的块设备，那为什么还要安装呢？答案是在安装之前可访问的只是这个设备，通常是作为一个线性的无结构的字节流来访问的，称为“原始设备”（raw device）；而设备上的文件系统则是不可访问的。经过安装以后，设备上的文件系统就成为可访问的了。

读者也许已经想到了一个问题，那就是：系统调用 mount()要求被安装的块设备在安装之前就是可访问的，那根设备怎么办？在安装根设备之前，系统中只有一个“/”节点，根本就不存在可访问的块设备啊。是的，根设备不能通过系统调用 mount()来安装。事实上，根据情况的不同，内核中有三个函数是用于设备安装的，那就是 sys_mount()、mount_root()以及 kem_mount()。我们先来看 sys_mount()，这就是系统调用 mount()在内核中的实现，其代码在 fs/super.c 中：

==================== fs/super.c 1421 1460 ====================

```
asmlinkage long sys_mount(char * dev_name, char * dir_name, char * type,
                unsigned long flags, void * data)
{
    int retval;
    unsigned long data_page;
    unsigned long type_page;
    unsigned long dev_page;
    char *dir_page;

    retval = copy_mount_options (type, &type_page);
    if (retval < 0)
        return retval;

    dir_page = getname(dir_name);
    retval = PTR_ERR(dir_page);
    if (IS_ERR(dir_page))
        goto out1;

    retval = copy_mount_options (dev_name, &dev_page);
    if (retval < 0)
        goto out2;

    retval = copy_mount_options (data, &data_page);
    if (retval < 0)
        goto out3;

    lock_kernel();
    retval = do_mount((char*)dev_page, dir_page, (char*)type_page,
                flags, (void*)data_page);
    unlock_kernel();
    free_page(data_page);

out3:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    free_page(dev_page);
out2:
    putname(dir_page);
out1:
    free_page(type_page);
    return retval;
}
```

参数 dev_name 为待安装设备的路径名；dir_name 则是安装点（空闲目录节点）的路径名；type 是表示文件系统类型（即格式）的字符串，如“ext2”、“iso9660”等。此外，flags 为安装模式，有关的标志位定义于 include/1inux/fs.h：

==================== include/linux/fs.h 96 120 ====================

```
/*
 * These are the fs-independent mount-flags: up to 32 flags are supported
 */
#define MS_RDONLY   1   /* Mount read-only */
#define MS_NOSUID   2   /* Ignore suid and sgid bits */
#define MS_NODEV    4   /* Disallow access to device special files */
#define MS_NOEXEC   8   /* Disallow program execution */
#define MS_SYNCHRONOUS 16   /* Writes are synced at once */
#define MS_REMOUNT 32   /* Alter flags of a mounted FS */
#define MS_MANDLOCK    64   /* Allow mandatory locks on an FS */
#define MS_NOATIME 1024 /* Do not update access times. */
#define MS_NODIRATIME  2048 /* Do not update directory access times */
#define MS_BIND        4096

/*
 * Flags that can be altered by MS_REMOUNT
 */
#define MS_RMT_MASK    (MS_RDONLY|MS_NOSUID|MS_NODEV|MS_NOEXEC|\
              MS_SYNCHRONOUS|MS_MANDLOCK|MS_NOATIME|MS_NODIRATIME)

/*
 * Magic mount flag number. Has to be or-ed to the flag values.
 */
#define MS_MGC_VAL 0xC0ED0000   /* magic flag number to indicate "new" flags */
#define MS_MGC_MSK 0xffff0000   /* magic flag number mask */
```

例如，如果 MS_NOSUID 标志为 1，而整个系统中所有可执行文件的 suid 标志位就都不起作用了。但是，正如原作者的注释所说，这些标志化并不是对所有文件系统都有效的。所有的标志位都在低 16 位中，而高 16 位则用作“magic_number”。

最后，指针 data 指向用于安装的附加信息，由不同文件系统的驱动程序自行加以解释，所以其类型为 void 指针。

代码中通过 getname()和 copy_mount_options()将字符串形式或结构形式的参数值从用户空间复制到系统空间。这些参数值的长度均以一个页面为限，但是 getname()在复制时遇到字符串结尾符“\0”就停止，并返回指向该字符串的指针；而 copy_mount_options()则拷贝整个页面（确切地说是

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

PAGE_SIZE-1 个字节），并且返回页面的起始地址。然后，就是这个操作的主体 do_mount()了。我们分段来看（super.c）：

==================== fs/super.c 1300 1343 ====================

[sys_mount()>do_mount()]

```
1300 /*
1301  * Flags is a 16-bit value that allows up to 16 non-fs dependent flags to
1302  * be given to the mount() call (ie: read-only, no-dev, no-suid etc).
1303  *
1304  * data is a (void *) that can point to any structure up to
1305  * PAGE_SIZE-1 bytes, which can contain arbitrary fs-dependent
1306  * information (or be NULL).
1307  *
1308  * NOTE! As pre-0.97 versions of mount() didn't use this setup, the
1309  * flags used to have a special 16-bit magic number in the high word:
1310  * 0xC0ED. If this magic number is present, the high word is discarded.
1311  */
1312 long do_mount(char * dev_name, char * dir_name, char *type_page,
1313               unsigned long flags, void *data_page)
1314 {
1315      struct file_system_type * fstype;
1316      struct nameidata nd;
1317      struct vfsmount *mnt = NULL;
1318      struct super_block *sb;
1319      int retval = 0;
1320
1321      /* Discard magic */
1322      if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
1323           flags &= ~MS_MGC_MSK;
1324
1325      /* Basic sanity checks */
1326
1327      if (!dir_name || !*dir_name || !memchr(dir_name, 0, PAGE_SIZE))
1328           return -EINVAL;
1329      if (dev_name && !memchr(dev_name, 0, PAGE_SIZE))
1330           return -EINVAL;
1331
1332      /* OK, looks good, now let's see what do they want */
1333
1334      /* just change the flags? - capabilities are checked in do_remount() */
1335      if (flags & MS_REMOUNT)
1336           return do_remount(dir_name, flags & ~MS_REMOUNT,
1337                          (char *) data_page);
1338
1339      /* "mount --bind"? Equivalent to older "mount -t bind" */
1340      /* No capabilities? What if users do thousands of these? */
1341      if (flags & MS_BIND)
1342           return do_loopback(dev_name, dir_name);
1343
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

首先是对参数的检验。例如对于安装节点名就要求指针 dir_name 不为 0，并且字符串的第一个字符不为 0，即不是空字符串，并且字符串的长度不超过一个页面。这里的 memchr()在指定长度的缓冲区中寻找指定的字符（这里是 0），如果找不到就返回 0。对设备名 dev_name 的检验很有趣：如果 dev_name 为非 0，则字符串的长度不得大于一个页面（实际上 copy_mount_options()保证了这一点，因为它拷贝 PAGE_SIZE-1 个字节），可是 dev_name 为 0 却是允许的。这似乎不可思议，下面读者将会看到，在特殊情况下这确实是允许的。

如果调用参数中的 MS_REMOUNT 标志位为 1，就表示所要求的只是改变一个原已安装的设备的安装方式。例如，原来是按"只读"方式来安装的，而现在要改为"可写"方式；或者原来的 MS_NOSUID 标志位为 0，而现在要改变成 1，等等。所以这种操作称为"重安装"。函数 do_remount()的代码也在 super.c 中，读者可以在阅读了 do_mount()的"主流"以后回过来自己读一下这个"支流"的代码。

另一个分支是对特殊设备如/dev/loopback 等"回接"设备的处理。这种设备是特殊的，其实并不是一种设备，而是一种机制。从系统的角度来看，它似乎是一种设备，但实际上它只是提供了一条" loopback"（回接）到某个可访问普通文件或块设备的手段。举例来说，系统的管理人员可以通过实用程序 losetup，实际上是系统调用 ioctl()，建立起/dev/loop0 与一个普通文件/blkfile 之间的联系，或者说将/dev/loop0"回接"到/blkfile，从而将这个文件当作一个块设备来使用：

losetup -e des /dev/loop0 /blkfile

这里的可选项-e des 表示在通过/dev/loop0 读写作为虚拟块设备的/blkfile 时要对内容加密，而加密的算法则为 DES（一种加密 / 解密标准）。也可以使用比较简单的加密算法 XOR，此时可选项即为" -e xor"。如果不加密就不用-e 可选项。回接以后，通过/dev/loop0 访问的文件/blkfile 就作为一个"块设备 "来使用了，所以也要加以格式化：

mkfs -t ext2 /dev/loop0 100

参数-t ext2 表示按 Ext2 格式化，也可以改用其它文件系统的格式。参数 100 表示该设备的大小为 100 个记录块。当然，文件/blkfile 原来的大小要足够，并且其原来的内容就丢失了，所以一般可以先建立起一个足够大的空文件：

dd if=/dev/zero of=/blkfile bs=1k count=100

回接的对象并不非得是一个普通文件，也可以是一个常规的块设备文件如/dev/hda2 等。但是，以普通文件作为回接对象给我们提供了将它格式化成一个文件系统并加以安装的手段。我们在回接时采用了加密，所以格式化以后的文件系统映象是加了密的，然后，就可以把这个虚拟的块设备安装到文件系统中了：

mount -t ext2 /dev/loop0 /nmt

从此以后，就跟一般已安装的子系统一样了，只是在我们这个例子中对这个子系统的读 / 写都加了密。

回接的对象还可以是一个已经安装的块设备。例如，/dev/hda1 已经安装在根节点/上，我们仍可以把它作为回接的对象。此时当然不能再加密，也不能再格式化了，但是还可以通过/dev/loop0 再安装一次（在另外一个节点上），例如把它安装成"只读"方式。如果回忆一下，一个进程（例如某种网络服务进程）可以通过系统调用设置自己的"根"目录，就不难想像这种"回接"设备对子系统安全性可能有用处了。通常在/dev 目录中有/dev/loop0 和/dev/loop1 两个回接设备文件，需要的话可以通过 mknod 再创建，其主设备号为 7。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

对通过回接设备的安装，以前在 mount 命令行中有个“ -o loop”可选项，现在则改成将命令行中的文件类型加上一种“ bind”，即“ -t bind”，表示所安装的设备是个“捆绑”到另一个对象上的回接设备。所以，如果 flags 中的 MS_BIND 标志位为 1（见代码中的第 1341 行），就调用 do_loopback()来完成回接设备的安装。我们暂且跳过它继续往下读（super.c）。

==================== fs/super.c 1344 1384 ====================

[sys_mount()>do_mount()]

```
1344     /* For the rest we need the type */
1345
1346     if (!type_page || !memchr(type_page, 0, PAGE_SIZE))
1347          return -EINVAL;
1348
1349 #if 0     /* Can be deleted again. Introduced in patch-2.3.99-pre6 */
1350     /* loopback mount? This is special - requires fewer capabilities */
1351     if (strcmp(type_page, "bind")==0)
1352          return do_loopback(dev_name, dir_name);
1353 #endif
1354
1355     /* for the rest we _really_ need capabilities... */
1356     if (!capable(CAP_SYS_ADMIN))
1357          return -EPERM;
1358
1359     /* ... filesystem driver... */
1360     fstype = get_fs_type(type_page);
1361     if (!fstype)
1362          return -ENODEV;
1363
1364     /* ... and mountpoint. Do the lookup first to force automounting. */
1365     if (path_init(dir_name,
1366               LOOKUP_FOLLOW|LOOKUP_POSITIVE|LOOKUP_DIRECTORY, &nd))
1367          retval = path_walk(dir_name, &nd);
1368     if (retval)
1369          goto fs_out;
1370
1371     /* get superblock, locks mount_sem on success */
1372     if (fstype->fs_flags & FS_NOMOUNT)
1373          sb = ERR_PTR(-EINVAL);
1374     else if (fstype->fs_flags & FS_REQUIRES_DEV)
1375          sb = get_sb_bdev(fstype, dev_name, flags, data_page);
1376     else if (fstype->fs_flags & FS_SINGLE)
1377          sb = get_sb_single(fstype, flags, data_page);
1378     else
1379          sb = get_sb_nodev(fstype, flags, data_page);
1380
1381     retval = PTR_ERR(sb);
1382     if (IS_ERR(sb))
1383          goto dput_out;
1384
```

进一步的操作需要系统管理员的权限，所以先检查当前进程是否具有此项授权。一般超级用户进程都是有这种授权的。

系统支持的每一种文件系统都有一个 file_system_type 数据结构，定义于 include/linux/fs.h：

==================== include/linux/fs.h 839 846 ====================

```
struct file_system_type {
	const char *name;
	int fs_flags;
	struct super_block *(*read_super) (struct super_block *, void *, int);
	struct module *owner;
	struct vfsmount *kern_mnt; /* For kernel mount, if it's FS_SINGLE fs */
	struct file_system_type * next;
};
```

结构中的 fs_flags 指明了具体文件系统的一些特性，有关的标志位定义见文件 fs.h：

==================== include/linux/fs.h 79 95 ====================

```
/* public flags for file_system_type */
#define FS_REQUIRES_DEV 1
#define FS_NO_DCACHE	2 /* Only dcache the necessary things. */
#define FS_NO_PRELIM	4 /* prevent preloading of dentries, even if
			   * FS_NO_DCACHE is not set.
			   */
#define FS_SINGLE	8 /*
			   * Filesystem that can have only one superblock;
			   * kernel-wide vfsmnt is placed in ->kern_mnt by
			   * kern_mount() which must be called _after_
			   * register_filesystem().
			   */
#define FS_NOMOUNT	16 /* Never mount from userland */
#define FS_LITTER	32 /* Keeps the tree in dcache */
#define FS_ODD_RENAME	32768	/* Temporary stuff; will go away as soon
				  * as nfs_rename() will be cleaned up
				  */
```

对这些标志位的意义和作用我们将随着代码解释的进展加以说明。

结构中有个函数指针 read_super，各种文件系统通过这个指针提供用来读入其超级块的函数，因为不同文件系统的超级块也是不同的。显然，这个数据结构也是从虚拟文件系统 VFS 进入具体文件系统的一个转接点。同时，每种文件系统还有个字符串形式的文件系统类型名。

安装文件系统时要说明文件系统的类型，例如系统命令 mount 就有个可选项"-t"，用于类型名。文件系统的类型名以字符串的形式复制到 type_page 中，现在就用来比对、寻找其 file_system_type 数据结构。

函数 get_fs_type()根据具体文件系统的类型名在内核中找到相应的 file_system_type 结构，有关的代码在 super.c 中：

==================== fs/super.c 262 279 ====================

[sys_mount()>do_mount()>get_fs_type()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
struct file_system_type *get_fs_type(const char *name)
{
    struct file_system_type *fs;

    read_lock(&file_systems_lock);
    fs = *(find_filesystem(name));
    if (fs && !try_inc_mod_count(fs->owner))
        fs = NULL;
    read_unlock(&file_systems_lock);
    if (!fs && (request_module(name) == 0)) {
        read_lock(&file_systems_lock);
        fs = *(find_filesystem(name));
        if (fs && !try_inc_mod_count(fs->owner))
            fs = NULL;
        read_unlock(&file_systems_lock);
    }
    return fs;
}
```

==================== fs/super.c 94 101 ====================

[sys_mount()>do_mount()>get_fs_type()>find_filesystem()]

```
static struct file_system_type **find_filesystem(const char *name)
{
    struct file_system_type **p;
    for (p=&file_systems; *p; p=&(*p)->next)
        if (strcmp((*p)->name,name) == 0)
            break;
    return p;
}
```

内核中有一个 file_system_type 结构队列，叫做 file_systems，队列中的每个数据结构都代表着一种文件系统。系统初始化时将内核支持的各种文件系统的 file_system_type 数据结构通过一个函数 register_filesystem()挂入这个队列，这个过程称为文件系统的注册。除此之外，对有些文件系统的支持可以通过“可安装模块”的方式来实现。在装入这些模块时，也会将相应的数据结构注册挂入该队列中。

函数 find_filesystem()则扫描 file_systems 队列，找到所需文件系统类型的数据结构。在 file_system_type 结构中有一个指针 owner，如果结构所代表的文件系统类型是通过可安装模块实现的，则该指针指向代表着具体模块的 module 结构。找到了 file_system_type 结构以后，要调用 try_inc_mod_count()看看该文件系统是否由可安装模块实现，是的话就要递增相应 module 结构中的共享计数，因为现在这个模块多了一个使用者。

要是在 file_systems 队列中找不到所需的文件系统类型怎么办呢？那就通过 request_module()试试能否（在已安装的文件系统中）找到用来实现所需文件系统类型的可安装模块，并将其装入内核；如果成功的话就再去 file_systems 队列中找一遍。如果装入所需的可安装模块失败，或者装入以后还是找不到相应的 file_system_type 结构，那就说明 Linux 系统不支持所要求的文件系统类型。有关模块的装入可参考“设备驱动”一章。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

同到 do_mount()的代码中。找到了给定文件系统类型的数据结构以后，就要寻找代表安装点的 dentry 数据结构了。通过 path_init()和 path_walk()寻找目标节点的过程以前已经讲过，就不重复了。找到了安装点的 dentry 结构（在 nameidata 结构 nd 中有个 dentry 指针）以后，要把待安装设备的“超级块”读进来并根据超级块中的信息在内存中建立起相应的 super_block 数据结构。但是，这里因具体文件系统的不同而有几种情形要区别对待：

(1) 有些虚拟的文件系统（如 pipe、共享内存区等），要由内核通过 kern_mount()安装，而根本不允许由用户进程通过系统调用 mount()来安装。这样的文件系统类型在其 fs_flag 中的 FS_NOMOUNT 标志位为 1。虚拟文件系统类型的“设备”其实没有超级块，所以只是按特定的内容初始化，或者说生成一个 super_block 结构。对于这种文件系统类型，系统调用 mount()时应出错返回。

(2) 一般的文件系统类型要求有物理的设备作为其物质基础，在其 fs_flags 中的 FS_REQUIRES_DEV 标志位为 1，这些就是“正常”的文件系统类型，如 ext2、minix、ufs 等等。对于这些文件系统类型，通过 get_sb_bdev()从待安装设备上读入其超级块。

(3) 有些虚拟文件系统在安装了同类型的的第一个“设备”，从而创建了超级块的 super_block 数据结构以后，再安装同一类型中的其它设备时就共享已经存在的 super_block 结构，而不再有其自己的超级块结构。此时相应 file_system_type 结构的 fs_flags 中的 FS_SINGLE 标志位为 1，表示整个文件系统类型只有一个超级块，而不像一般的文件系统类型那样每个具体的设备上都有一个超级块。

(4) 还有些文件系统类型的 fs_flags 中的 FS_NOMOUNT 标志位、FS_REQUIRE_DEV 标志位以及 FS_SINGLE 标志位全都为 0，所以不属于上述三种情形中的任何一种。这些所谓“文件系统 ”其实也是虚拟的，通常只是用来实现某种机制或者规程，所以根本就没有“设备”。对于这样的“文件系统类型”都是通过 get_sb_nodev()来生成一个 super_block 结构的。

总之，每种文件系统类型都有个 file_system_type 结构，而结构中的 fs_flags 则由各种标志位组成，这些标志位表明了具体文件系统类型的特性，也决定着这种文件系统的安装过程。内核代码中提供了两个用来建立 file_system_type 数据结构的宏操作，其定义在 fs.h 中：

==================== include/linux/fs.h 848 857 ====================

```
#define DECLARE_FSTYPE(var,type,read,flags) \
struct file_system_type var = { \
	name:		type, \
	read_super:	read, \
	fs_flags:flags, \
	owner:		THIS_MODULE, \
}

#define DECLARE_FSTYPE_DEV(var,type,read) \
	DECLARE_FSTYPE(var,type,read,FS_REQUIRES_DEV)
```

一般常规的文件系统类型都通过 DECLARE_FSTYPE_DEV 建立其数据结构，因为它们的 FS_REQUIRE_DEV 标志位均为 1，而其它标志位为 0，例如 fs/ext2/super.c 中的 ext2_fs_type：

==================== fs/ext2/super.c 786 786 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static DECLARE_FSTYPE_DEV(ext2_fs_type, "ext2", ext2_read_super);
```

再如 fs/msdos/msdosfs_syms.c 中的 msdos_fs_type：

==================== fs/msdos/msdosfs_syms.c 29 29 ====================

```
static DECLARE_FSTYPE_DEV(msdos_fs_type, "msdos", msdos_read_super);
```

相比之下，特殊的、虚拟的文件系统类型则大多直接通过 DECLARE_FSTYPE 建立起数据结构，因为它们的 fs_flags 是特殊的，例如 fs/pipe.c 中的 pipe_fs_type：

==================== fs/pipe.c 632 633 ====================

```
static DECLARE_FSTYPE(pipe_fs_type, "pipefs", pipefs_read_super,
    FS_NOMOUNT|FS_SINGLE);
```

以及 fs/ramfs/inode.c 中的 ramfs_fs_type：

==================== fs/ramfs/inode.c 336 336 ====================

```
static DECLARE_FSTYPE(ramfs_fs_type, "ramfs", ramfs_read_super, FS_LITTER);
```

以后读者会看到，flags 中的 FS_SINGLE 标志位有着很重要的作用。我们在这里只关心常规文件系统的安装，所以只阅读 get_sb_bdev()的代码，以后我们会结合其它章节，如进程间通信和设备驱动，再来阅读 get_sb_single()等函数的代码。顺便提一下，这里 get_sb_single()和 get_sb_nodev()都不使用参数 dev_name，所以它可以是 NULL。这个函数的代码也在 fs/super.c 中，我们分段阅读。

==================== fs/super.c 785 808 ====================

[sys_mount()>do_mount()>get_sb_bdev()]

```
static struct super_block *get_sb_bdev(struct file_system_type *fs_type,
    char *dev_name, int flags, void * data)
{
    struct inode *inode;
    struct block_device *bdev;
    struct block_device_operations *bdops;
    struct super_block * sb;
    struct nameidata nd;
    kdev_t dev;
    int error = 0;
    /* What device it is? */
    if (!dev_name || !*dev_name)
        return ERR_PTR(-EINVAL);
    if (path_init(dev_name, LOOKUP_FOLLOW|LOOKUP_POSITIVE, &nd))
        error = path_walk(dev_name, &nd);
    if (error)
        return ERR_PTR(error);
    inode = nd.dentry->d_inode;
    error = -ENOTBLK;
    if (!S_ISBLK(inode->i_mode))
        goto out;
    error = -EACCES;
    if (IS_NODEV(inode))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        goto out;
```

对于常规的文件系统，参数 dev_name 必须是一个有效的路径名。同样，这里也是通过 path_init() 和 path_walk()找到目标节点，即相应设备文件的 dentry 结构以及 inode 结构。当然，找到的 inode 结构必须是代表着一个块设备，其 i_mode 中的 S_IFBLK 标志位必须为 1，否则就错了。宏操作 S_ISBLK() 定义于 include/linux/stat.h：

==================== include/linux/stat.h 28 28 ====================

```
#define S_ISBLK(m) (((m) & S_IFMT) == S_IFBLK)
```

设备文件的 inode 结构是在 path_walk()中根据从已经安装的磁盘上（或其它已安装的文件系统中）读入的索引节点建立的。列于 Ext2 文件系统，我们在“从路径名到目标节点”一节中阅读 path_walk() 的代码时曾在它所辗转调用的 ext2_read_inode()中看到这么一段代码：

==================== fs/ext2/inode.c 1059 1078 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()>ext2_read_inode()]

```
    if (inode->i_ino == EXT2_ACL_IDX_INO ||
        inode->i_ino == EXT2_ACL_DATA_INO)
        /* Nothing to do */ ;
    else if (S_ISREG(inode->i_mode)) {
        inode->i_op = &ext2_file_inode_operations;
        inode->i_fop = &ext2_file_operations;
        inode->i_mapping->a_ops = &ext2_aops;
    } else if (S_ISDIR(inode->i_mode)) {
        inode->i_op = &ext2_dir_inode_operations;
        inode->i_fop = &ext2_dir_operations;
    } else if (S_ISLNK(inode->i_mode)) {
        if (!inode->i_blocks)
            inode->i_op = &ext2_fast_symlink_inode_operations;
        else {
            inode->i_op = &page_symlink_inode_operations;
            inode->i_mapping->a_ops = &ext2_aops;
        }
    } else
        init_special_inode(inode, inode->i_mode,
                   le32_to_cpu(raw_inode->i_block[0]));
```

由于设备文件既不是常规文件，也不是目录，更不是符号连接，所以必然会调用 init_special_inode()，其代码在 fs/devices.c 中：

==================== fs/devices.c 200 216 ====================

[path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()>ext2_read_inode()>init_special_inode()]

```
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = to_kdev_t(rdev);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = to_kdev_t(rdev);
        inode->i_bdev = bdget(rdev);
    } else if (S_ISFIFO(mode))
        inode->i_fop = &def_fifo_fops;
    else if (S_ISSOCK(mode))
        inode->i_fop = &bad_sock_fops;
    else
        printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
}
```

以前说过，在 inode 数据结构中有两个设备号。一个是索引节点所在设备的号码 i_dev，另一个是索引节点所代表的设备的号码 i_rdev。可是，如果看一下存储在设备上的索引节点 ext2_inode 数据结构，就可以发现里面一个专门用于设备号的字段也没有。首先，既然索引节点存储在某个设备上，当然就不需要再在里面说明存储在哪个设备上了。再说，一个索引节点如果代表着一个设备，那就不需要记录跟文件的物理信息有关的数据了，从而可以利用这些空间来记录所代表设备的设备号。事实上，当索引节点代表着设备时，其 ext2_inode 数据结构中的数组 i_block[]空着没用，所以就将 i_block[0]用于设备号。这个设备号在这里的 init_special_node()中经过 to_kdev_t()加以格式转换以后就变成 inode 结构中的 i_rdev。此外，对于块设备还要使 inode 结构中的指针 i_bdev 指向一个 block_device 结构。具体的数据结构由 bdget()根据设备号寻找或创建，详见“设备驱动”一章中有关的内容。

回到 get_sb_bdev()的代码中（fs/super.c）：

==================== fs/super.c 809 847 ====================

[sys_mount()>do_mount()>get_sb_bdev()]

```
    bdev = inode->i_bdev;
    bdops = devfs_get_ops ( devfs_get_handle_from_inode (inode) );
    if (bdops) bdev->bd_op = bdops;
    /* Done with lookups, semaphore down */
    down(&mount_sem);
    dev = to_kdev_t(bdev->bd_dev);
    sb = get_super(dev);
    if (sb) {
        if (fs_type == sb->s_type &&
            ((flags ^ sb->s_flags) & MS_RDONLY) == 0) {
             path_release(&nd);
             return sb;
        }
    } else {
        mode_t mode = FMODE_READ; /* we always need it ;-) */
        if (!(flags & MS_RDONLY))
             mode |= FMODE_WRITE;
        error = blkdev_get(bdev, mode, 0, BDEV_FS);
        if (error)
             goto out;
        check_disk_change(dev);
        error = -EACCES;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (!(flags & MS_RDONLY) && is_read_only(dev))
            goto out1;
        error = -EINVAL;
        sb = read_super(dev, bdev, fs_type, flags, data, 0);
        if (sb) {
            get_filesystem(fs_type);
            path_release(&nd);
            return sb;
        }
out1:
        blkdev_put(bdev, BDEV_FS);
    }
out:
    path_release(&nd);
    up(&mount_sem);
    return ERR_PTR(error);
}
```

在 block_device 结构中有个指针 bd_op，指向一个 block_device_operations 数据结构，这就是块设备驱动程序的函数跳转表。所以，我们可以把 block_device 结构比喻为块设备驱动“总线”，而使其指针 bd_op 指向某个具体的 block_device_operations 数据结构，就好像是将一块“接口卡”插入了总线的插槽，这跟 VFS 与具体文件系统的关系是一样的。

那么，这里要把什么样的“接口卡”插到总线上去呢？原来，在 Linux 的设备驱动方面正在进行着一项称为“ devfs”的改革。传统的/dev 目录是一种“平面”结构而不像其它目录那样是树状结构。每一个设备都有个“主设备号”和一个“次设备号”。每当要在/dev 中建立一个节点（即设备文件）时就要将主、次设备号合成一个单一的“设备号”，再通过系统调用 mknod()来建立，传统的主、次设备号都是 8 位的，所以每种设备最多只能有 255 个。随着技术的发展，这个限制开始成为问题了。所以 Linux 内核已经开始使用 16 位的主、次设备号。可是，另有一派意见认为，/dev 的这种平面结构和主、次设备号的使用根本就应该改革。也就是说，把/dev 改成树状结构，这样一来路径名就可以惟一地确定一个设备的类型和序号，例如/dev/hda/1，这样就可以把主、次设备号隐藏在路径名的背后，不需要在用户界面上用什么主设备号、次设备号了。目前这项改革正在进行中，对有些设备（如软盘、磁带等）的支持已开始使用这种新的方案。但是，内核必须同时支持新、旧两种方案，这里对 devfs 和 devfs_get_handle_from_inode()就是出于对 devfs 的考虑。目前（以及在未来相当一段时期内），对多数块设备的支持还会沿用传统的模式，如果尚不支持 devfs 则这两个函数都返回 NULL 而不起作用，相当于让插槽暂时空着。我们在“设备驱动”一章中还要回到 devfs 这个话题上。另一方面，由于在内核中已经开始使用 16 位的主、次设备号，而在大多数文件系统中都还是 8 位的，所以要通过 to_kdev_t()加以转换。

完成了上面的这些准备以后，现在要进行实质性的工作，就是找到或建立待安装设备的 super_block 数据结构了。首先还是在内核中寻找，内核中维持着一个 super_block 数据结构的队列 super_blocks，所有的 super_block 结构，包括空闲的，都通过结构中的一个队列头 s_list 链入到这个队列中。寻找时就通过 get_super()从队列中寻找，其代码在 fs/super.c 中：

==================== fs/super.c 631 656 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_mount()>do_mount()>get_sb_bdev()>get_super()]

```
/**
 *  get_super -   get the superblock of a device
 *  @dev: device to get the superblock for
 *
 *  Scans the superblock list and finds the superblock of the file system
 *  mounted on the device given. %NULL is returned if no match is found.
 */

struct super_block * get_super(kdev_t dev)
{
	struct super_block * s;

	if (!dev)
		return NULL;
restart:
	s = sb_entry(super_blocks.next);
	while (s != sb_entry(&super_blocks))
		if (s->s_dev == dev) {
			wait_on_super(s);
			if (s->s_dev == dev)
				return s;
			goto restart;
		} else
			s = sb_entry(s->s_list.next);
	return NULL;
}
```

这里的 sb_entry()是个宏操作，定义于 include/linux/fs.h：

==================== include/linux/fs.h 664 664 ====================

```
#define sb_entry(list) list_entry((list), struct super_block, s_list)
```

读者也许会问，这是否意味着同一个块设备可以安装多次？答案是可以的，例如我们在前面曾经讲到通过“回接设备”进行的安装，那就是同一设备的多次安装。

然而，在大多数情况下 get_super()实际上都会失败，因而得从设备读入其超级块并在内存中建立起该设备的 super_block 数据结构。为了这个目的，先得要“打开”这个设备文件，这是由 blkdev_get()完成的，其代码在 fs/block_dev.c 中：

==================== fs/block_dev.c 606 642 ====================

[sys_mount()>do_mount()>get_sb_bdev()>blkdev_get()]

```
int blkdev_get(struct block_device *bdev, mode_t mode, unsigned flags, int kind)
{
	int ret = -ENODEV;
	kdev_t rdev = to_kdev_t(bdev->bd_dev); /* this should become bdev */
	down(&bdev->bd_sem);
	if (!bdev->bd_op)
		bdev->bd_op = get_blkfops(MAJOR(rdev));
	if (bdev->bd_op) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /*
         * This crockload is due to bad choice of ->open() type.
         * It will go away.
         * For now, block device ->open() routine must _not_
         * examine anything in 'inode' argument except ->i_rdev.
         */
        struct file fake_file = {};
        struct dentry fake_dentry = {};
        struct inode *fake_inode = get_empty_inode();
        ret = -ENOMEM;
        if (fake_inode) {
            fake_file.f_mode = mode;
            fake_file.f_flags = flags;
            fake_file.f_dentry = &fake_dentry;
            fake_dentry.d_inode = fake_inode;
            fake_inode->i_rdev = rdev;
            ret = 0;
            if (bdev->bd_op->open)
                ret = bdev->bd_op->open(fake_inode, &fake_file);
            if (!ret)
                atomic_inc(&bdev->bd_openers);
            else if (!atomic_read(&bdev->bd_openers))
                bdev->bd_op = NULL;
            iput(fake_inode);
        }
    }
    up(&bdev->bd_sem);
    return ret;
}
```

由于 block_device 结构中的 bd_dev 有可能还在使用 8 位的主、次设备号，或者说 16 位的设备号，这里先通过 to_kdev_t()将它们换成 16 位（或者说 32 位的设备号）。前面讲过，block_device 结构中的指针 bd_op 指向一个 blocK_device_operations 数据结构。对于 devfs 的设备这个指针已经在前面设置好了，而对于传统的块设备则这个指针尚未设置，暂时还空着，所以要通过 get_blkfops()根据设备的主设备号来设置这个指针。函数 get_blkfops()的代码也在 fs/block_dev.c 中：

==================== fs/block_dev.c 487 507 ====================

[sys_mount()>do_mount()>get_sb_bdev()>blkdev_get()>get_blkfops()]

```
/*
    Return the function table of a device.
    Load the driver if needed.
*/
const struct block_device_operations * get_blkfops(unsigned int major)
{
    const struct block_device_operations *ret = NULL;

    /* major 0 is used for non-device mounts */
    if (major && major < MAX_BLKDEV) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#ifdef CONFIG_KMOD
          if (!blkdevs[major].bdops) {
               char name[20];
               sprintf(name, "block-major-%d", major);
               request_module(name);
          }
#endif
          ret = blkdevs[major].bdops;
     }
     return ret;
}
```

内核中设置了一个以主设备号为下标的结构数组 blkdevs[]，用来保存指向各种块设备的 block_device_operations 结构的指针：

==================== fs/block_dev.c 468 471 ====================

```
static struct {
     const char *name;
     struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];
```

系统在初始化时将所支持的各种块设备的block_device_operations结构指针填入该数组中的相应元素中。以可安装模块实现的设备驱动程序则在装入模块时才设置相应的指针。所以，如果相应表项的 bdops 指针为 0，则表明该设备可能是以可安装模块实现的，但是尚未装入，因此要调用 request_module()将其装入。在正常情况下，当从 get_blkfops()返回时指针 bdev->bd->op 已经设置好了，就好像“接口卡”已经插入了“总线”。

为了打开设备，还需要使用几个临时的数据结构，包括 file 结构、dentry 结构以及 inode 结构。这里要指出，我们现在要打开的是作为文件的设备本身，而不是这个设备在文件系统中的代表如“/dev/hda1”等节点，那早已经打开了，要不然就无从知道其主设备号和次设备号了。打开设备的操作是通过由具体设备类型的 block_device_operations 结构中的函数指针 open 提供的。就一般的 ide 磁盘而言，其数据结构为 bd_fops，而相应的函数指针则指向 bd_open()。我们在这里就不深入到打开设备的过程中去了，读者可参阅打开文件及设备驱动等有关章节。

打开了设备，blkdev_get()也就完成了。回到 get_sb_bdev()中，还要作一些检查。有些设备的介质是活动的，可以由用户替换的（例如软盘），对于这样的设备要检查一下其介质是否已经变动了（如果原来已经安装的话）。我们在这里只关心固定介质磁盘，所以就不深入到 check_disk_change()的代码中去了。有兴趣或有需要的读者可以自己阅读。最后，还有一项检查，那就是如果安装的模式不是“只读”而所欲安装的设备却已经设置成“只读”，那就不能安装了。

打开了具体的设备以后，就要通过 read_super()从设备上读入超级块并在内存中建立起 super_block 结构了，其代码还是在 fs/super.c 中：

==================== fs/super.c 721 755 ====================

[sys_mount()>do_mount()>get_sb_bdev()>read_super()]

```
static struct super_block * read_super(kdev_t dev, struct block_device *bdev,
                         struct file_system_type *type, int flags,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                              void *data, int silent)
{
        struct super_block * s;
        s = get_empty_super();
        if (!s)
                goto out;
        s->s_dev = dev;
        s->s_bdev = bdev;
        s->s_flags = flags;
        s->s_dirt = 0;
        sema_init(&s->s_vfs_rename_sem,1);
        sema_init(&s->s_nfsd_free_path_sem,1);
        s->s_type = type;
        sema_init(&s->s_dquot.dqio_sem, 1);
        sema_init(&s->s_dquot.dqoff_sem, 1);
        s->s_dquot.flags = 0;
        lock_super(s);
        if (!type->read_super(s, data, silent))
                goto out_fail;
        unlock_super(s);
        /* tell bdcache that we are going to keep this one */
        if (bdev)
                atomic_inc(&bdev->bd_count);
out:
        return s;

out_fail:
        s->s_dev = 0;
        s->s_bdev = 0;
        s->s_type = NULL;
        unlock_super(s);
        return NULL;
}
```

先从 super_blocks 队列中找到一个空闲的 super_block 结构，进行一些简单的初始化以后就要根据具体设备上的文件系统类型读入超级块。如前所述，在代表着具体文件系统类型的 file_system_type 数据结构中有个函数指针 read_super()指向具体的函数。对于 Ext2 文件系统，其数据结构为 ext2_fs_type，而相应的函数指针则指向 ext2_read_super()。函数 ext2_read_super()相当大，有 250 多行，而它的逻辑和过程则相对独立，所以我们把它暂时放一下，以后再来读它的代码，现在先继续往下看。

从设备上读入超级块并设置好 super_block 结构以后，get_sb_bdev()的工作就完成了，只是返回前可能还需要递增用来实现此种文件系统类型的可安装模块的使用者计数：

==================== fs/super.c 81 86 ====================

[sys_mount()>do_mount()>get_sb_bdev()>get_filesystem()]

```
/* WARNING: This can be used only if we _already_ own a reference */
static void get_filesystem(struct file_system_type *fs)
{
        if (fs->owner)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        __MOD_INC_USE_COUNT(fs->owner);
}
```

此外，还要通过 path_release()释放在 path_walk()中占用的 dentry 结构（代表着待安装设备）和可能的 vfsmount 结构。

回到 do_mount()的代码中继续往下读（fs/super.c）。

==================== fs/super.c 1385 1419 ====================

[sys_mount()>do_mount()]

```
    /* Something was mounted here while we slept */
    while(d_mountpoint(nd.dentry) && follow_down(&nd.mnt, &nd.dentry))
        ;

    /* Refuse the same filesystem on the same mount point */
    retval = -EBUSY;
    if (nd.mnt && nd.mnt->mnt_sb == sb
           && nd.mnt->mnt_root == nd.dentry)
        goto fail;

    retval = -ENOENT;
    if (!nd.dentry->d_inode)
        goto fail;
    down(&nd.dentry->d_inode->i_zombie);
    if (!IS_DEADDIR(nd.dentry->d_inode)) {
        retval = -ENOMEM;
        mnt = add_vfsmnt(&nd, sb->s_root, dev_name);
    }
    up(&nd.dentry->d_inode->i_zombie);
    if (!mnt)
        goto fail;
    retval = 0;
unlock_out:
    up(&mount_sem);
dput_out:
    path_release(&nd);
fs_out:
    put_filesystem(fstype);
    return retval;

fail:
    if (list_empty(&sb->s_mounts))
        kill_super(sb, 0);
    goto unlock_out;
}
```

待安装设备的 super_block 结构已经解决了，这一边已经没有什么问题，现在要回过头来看安装点这一边了。前面，在处理待安装设备的超级块之前，已经通过 path_init()和 path_walk()找到了安装点的 dentry 结构、inode 结构以及 vfsmount 结构，通过局部量 nameidata 数据结构 nd 就可以访问到这些数据

结构。但是还有一种情况需要考虑。

首先，前面从设备上读入超级块的过程是个颇为漫长的过程，当前进程在等待从设备上读入的过程中几乎可肯定要进入睡眠，这样就可能会有另一个进程捷足先登抢先将另一个设备安装到了同一个安装点上。要知道是否发生了这种情况，可以通过 d_mountpoint()来检测（见文件 dcache.h）：

==================== include/linux/dcache.h 259 262 ====================

[sys_mount()>do_mount()>d_mountpoint()]

```
static __inline__ int d_mountpoint(struct dentry *dentry)
{
	return !list_empty(&dentry->d_vfsmnt);
}
```

如果代表着安装点的 dentry 结构中的 d_vfsmnt 队列非空，那就说明已经有设备安装在上面了。在这种情况下怎么办呢？我们从代码中看到其对策是调用 follow_down()前进到已安装设备上的根节点，并且要通过 while 循环进一步检测新的安装点，直到尽头，即前进到不再有设备安装的某个设备上的根节点为止。已安装设备的根目录下面一般都是有内容的，是否可以把一个设备安装在一个非空的目录节点上呢？可以的。这一点可能与人们的直觉和想像不同。但是，将一个设备安装到一个有内容的目录节点时，该节点就变成了一个纯粹的安装点，原来目录中的内容就变成不可访问了。当然，从管理的角度出发应该避免发生这种情况，但是就技术角度而言这是可以的。

==================== fs/namei.c 375 378 ====================

[sys_mount()>do_mount()>follow_down()]

```
int follow_down(struct vfsmount **mnt, struct dentry **dentry)
{
	return __follow_down(mnt,dentry);
}
```

这个函数只是将一个 inline 函数__follow_down()抽出来作为一个普通的函数(读者应该明白二者的区别)。以前，我们在 path_walk()的代码中也看到过对这个 inline 函数的引用。其代码在 fs/namei.c 中：

==================== fs/namei.c 352 373 ====================

```
static inline int __follow_down(struct vfsmount **mnt, struct dentry **dentry)
{
	struct list_head *p;
	spin_lock(&dcache_lock);
	p = (*dentry)->d_vfsmnt.next;
	while (p != &(*dentry)->d_vfsmnt) {
		struct vfsmount *tmp;
		tmp = list_entry(p, struct vfsmount, mnt_clash);
		if (tmp->mnt_parent == *mnt) {
			*mnt = mntget(tmp);
			spin_unlock(&dcache_lock);
			mntput(tmp->mnt_parent);
			/* tmp holds the mountpoint, so... */
			dput(*dentry);
			*dentry = dget(tmp->mnt_root);
			return 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }
        p = p->next;
    }
    spin_unlock(&dcache_lock);
    return 0;
}
```

把一个设备安装到一个目录节点时要用一个 vfsmount 数据结构作为“连接件”。该数据结构定义于 include/linux/mount.h：

==================== include/linux/mount.h 17 33 ====================

```
struct vfsmount
{
    struct dentry *mnt_mountpoint;   /* dentry of mountpoint */
    struct dentry *mnt_root;    /* root of the mounted tree */
    struct vfsmount *mnt_parent;     /* fs we are mounted on */
    struct list_head mnt_instances; /* other vfsmounts of the same fs */
    struct list_head mnt_clash; /* those who are mounted on (other */
                       /* instances) of the same dentry */
    struct super_block *mnt_sb; /* pointer to superblock */
    struct list_head mnt_mounts;     /* list of children, anchored here */
    struct list_head mnt_child; /* and going through their mnt_child */
    atomic_t mnt_count;
    int mnt_flags;
    char *mnt_devname;      /* Name of device e.g. /dev/dsk/hda1 */
    struct list_head mnt_list;
    uid_t mnt_owner;
};
```

结构中主要成分的作用为：

- 指针 mnt_mountpoint 指向安装点的 dentry 数据结构，而指针 mount_root 则指向所安装设备上根目录的 dentry 数据结构，在二者之间搭起一座桥梁。
- 可是，在 dentry 结构中却没有直接指向 vfsmount 数据结构的指针，而是有个队列头 d_vfsmount，这是因为安装点和设备之间是一对多的关系，在同一个安装点上可以安装多个设备。相应地，vfsmount 结构中也有个队列头 mnt_clash，通过它链入到安装点的 d_vfsmount 队列中。不过，从所安装设备上根目录的 dentry 数据结构出发却不能直接找到其 vfsmount 结构，而得要通过其 super_block 数据结构中转。
- 指针 mnt_sb 指向所安装设备的超级块的 super_block 数据结构。反之，在所安装设备的 super_block 数据结构中却并没有直接指向 vfsmount 数据结构的指针，而是有个队列头 s_mounts，因为设备与安装点之间也是一对多的关系，同一个设备可以安装到多个安装点上。相应地，vfsmount 结构中也有个队列头 mnt_instances，通过它链入到设备的 s_mounts 队列中。
- 指针 mnt_parent 指向安装点所在设备当初安装时的 vfsmount 数据结构，就是上一层的 vfsmount 数据结构。不过，在根设备或其它不存在上一层 vfsmount 数据结构的情况下，这个指针指向该数据结构本身。同时，vfsmount 数据结构中还有 mnt_child 和 mnt_mounts 两个队列头，只要上一层的 vfsmount 数据结构存在，就通过 mnt_child 链入上一层 vfsmount 结构的 mnt_mounts

PDF-XCHANGE Click to buy NOW! www.docu-track.com

队列中。这样，就形成一种设备安装的树形结构，从一个 vfsmount 结构的 mnt_mounts 队列开始可以找到所有直接或间接安装在这个设备上（的文件系统中）的其它设备。

- 此外，系统中还有个总的 vfsmount 结构队列 vfsmntlist，相应地 vfsmount 数据结构中还有个队列头 mnt_list。所有已安装设备的 vfsmount 结构都通过 mnt_list 链入 vfsmntlist 队列中。

所安装设备的 super_block 数据结构与作为“连接件”的 vfsmount 数据结构之间可以是一对多的关系，这容易理解，因为把同一物理设备安装到文件系统中不同的节点上，成为逻辑上相互独立的子树是很自然的事。可是，安装点的 dentry 结构与 vfsmount 结构之间也可以是一对多的关系，这就不容易理解了。很难想像怎么可以把多个设备安装到同一个节点上。其实，这二者是联系的，有了前者就会有后者。我们通过一个假想的情景来说明这个问题：假定有/dev/hda1、/dev/hda2、/dev/hda3、和/dev/hda4 四个设备（磁盘分区），/dev/hda1 为根设备（这四个设备文件节点都在/dev/hda1 上的/dev 目录下），并且，在/dev/hda1 的根目录下有两个空闲的目录节点/d11 和/d12，而在/dev/hda2 的根目录下则有个空闲的目录节点 d2。现在把/dev/hda2 分别安装到/d11 和/d12 上去，这当然是可以的。可是，这样一来就有了/d11/d2 和/d12/d2 两个路径通往同一个物理的目录节点。然后，把/dev/hda3 安装到/d11/d2 上，这样/d11/d2 就代表着/dev/hda3 了。可是/d12/d2 呢？显然，它应该还是空的，因为/d12 代表着一棵独立的子树。再把/dev/hda4 安装到/d12/d2 上，这当然也是允许的。好，现在/dev/hda2 上的目录节点 d2 就安装着两个设备了，从而有两个 vfsmount 数据结构在其 dentry 结构的 d_vfsmount 队列中。读者自然就会产生一个问题：这样，当沿着路径名搜索，发现 d2 是个安装点而要前进到所安装的设备上时，怎么知道到底是要前进到/dev/hda3 还是/dev/hda4 呢？显然这时候需要看路径名的“上下文”。具体地，要看是顺着/dev/hda2 的哪一次安装（/d11/d2 或/d12/d2）搜索下来的，而上一层的 vfsmount 数据结构实际上就代表着这个上下文。所以，在上面__follow_down()的代码中是一个 while 循环，它扫描 dentry 结构中的 d_vfsrnount 队列中的所有 vfsmount 数据结构，找出其中上一层 vfsmount 数据结构相符的那个“连接件”。

回到 do_mount()的代码中。安装点最终确定以后，剩下的就是把待安装设备的 super_block 数据结构与安装点的 dentry 数据结构联系在一起，即“安装”本身了，这是通过 add_vfsmnt()完成的，其代码在 fs/super.c 中：

==================== fs/super.c 281 360 ====================

[sys_mount()>do_mount()>add_vfsmnt()]

```
static LIST_HEAD(vfsmntlist);

/**
 *  add_vfsmnt - add a new mount node
 *  @nd: location of mountpoint or %NULL if we want a root node
 *  @root: root of (sub)tree to be mounted
 *  @dev_name: device name to show in /proc/mounts or %NULL (for "none").
 *
 *  This is VFS idea of mount. New node is allocated, bound to a tree
 *  we are mounting and optionally (OK, usually) registered as mounted
 *  on a given mountpoint. Returns a pointer to new node or %NULL in
 *  case of failure.
 *
 *  Potential reason for failure (aside of trivial lack of memory) is a
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   deleted mountpoint. Caller must hold ->i_zombie on mountpoint
 *   dentry (if any).
 *
 *   Node is marked as MNT_VISIBLE (visible in /proc/mounts) unless both
 *   @nd and @devname are %NULL. It works since we pass non-%NULL @devname
 *   when we are mounting root and kern_mount() filesystems are deviceless.
 *   If we will get a kern_mount() filesystem with nontrivial @devname we
 *   will have to pass the visibility flag explicitly, so if we will add
 *   support for such beasts we'll have to change prototype.
 */

static struct vfsmount *add_vfsmnt(struct nameidata *nd,
                  struct dentry *root,
                  const char *dev_name)
{
    struct vfsmount *mnt;
    struct super_block *sb = root->d_inode->i_sb;
    char *name;

    mnt = kmalloc(sizeof(struct vfsmount), GFP_KERNEL);
    if (!mnt)
        goto out;
    memset(mnt, 0, sizeof(struct vfsmount));

    if (nd || dev_name)
        mnt->mnt_flags = MNT_VISIBLE;

    /* It may be NULL, but who cares? */
    if (dev_name) {
        name = kmalloc(strlen(dev_name)+1, GFP_KERNEL);
        if (name) {
            strcpy(name, dev_name);
            mnt->mnt_devname = name;
        }
    }
    mnt->mnt_owner = current->uid;
    atomic_set(&mnt->mnt_count,1);
    mnt->mnt_sb = sb;

    spin_lock(&dcache_lock);
    if (nd && !IS_ROOT(nd->dentry) && d_unhashed(nd->dentry))
        goto fail;
    mnt->mnt_root = dget(root);
    mnt->mnt_mountpoint = nd ? dget(nd->dentry) : dget(root);
    mnt->mnt_parent = nd ? mntget(nd->mnt) : mnt;

    if (nd) {
        list_add(&mnt->mnt_child, &nd->mnt->mnt_mounts);
        list_add(&mnt->mnt_clash, &nd->dentry->d_vfsmnt);
```

```
	} else {
		INIT_LIST_HEAD(&mnt->mnt_child);
		INIT_LIST_HEAD(&mnt->mnt_clash);
	}
	INIT_LIST_HEAD(&mnt->mnt_mounts);
	list_add(&mnt->mnt_instances, &sb->s_mounts);
	list_add(&mnt->mnt_list, vfsmntlist.prev);
	spin_unlock(&dcache_lock);
out:
	return mnt;
fail:
	spin_unlock(&dcache_lock);
	if (mnt->mnt_devname)
		kfree(mnt->mnt_devname);
	kfree(mnt);
	return NULL;
}
```

至此，设备的安装就完成了。

看完了 do_mount()的主流，我们再来看看它的一个支流 do_loopback()。我们在前面谈论过“回接设备”，现在就来看它是怎样安装的。函数 do_loopback()的代码也在文件 fs/super.c 中：

==================== fs/super.c 1182 1228 ====================

[sys_mount()>do_mount()>do_loopback()]

```
/*
 * do loopback mount.
 */
static int do_loopback(char *old_name, char *new_name)
{
	struct nameidata old_nd, new_nd;
	int err = 0;
	if (!old_name || !*old_name)
		return -EINVAL;
	if (path_init(old_name, LOOKUP_POSITIVE, &old_nd))
		err = path_walk(old_name, &old_nd);
	if (err)
		goto out;
	if (path_init(new_name, LOOKUP_POSITIVE, &new_nd))
		err = path_walk(new_name, &new_nd);
	if (err)
		goto out1;
	err = mount_is_safe(&new_nd);
	if (err)
		goto out2;
	err = -EINVAL;
	if (S_ISDIR(new_nd.dentry->d_inode->i_mode) !=
	     S_ISDIR(old_nd.dentry->d_inode->i_mode))
		goto out2;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    err = -ENOMEM;
    if (old_nd.mnt->mnt_sb->s_type->fs_flags & FS_SINGLE)
        get_filesystem(old_nd.mnt->mnt_sb->s_type);

    down(&mount_sem);
    /* there we go */
    down(&new_nd.dentry->d_inode->i_zombie);
    if (IS_DEADDIR(new_nd.dentry->d_inode))
        err = -ENOENT;
    else if (add_vfsmnt(&new_nd, old_nd.dentry, old_nd.mnt->mnt_devname))
        err = 0;
    up(&new_nd.dentry->d_inode->i_zombie);
    up(&mount_sem);
    if (err && old_nd.mnt->mnt_sb->s_type->fs_flags & FS_SINGLE)
        put_filesystem(old_nd.mnt->mnt_sb->s_type);
out2:
    path_release(&new_nd);
out1:
    path_release(&old_nd);
out:
    return err;
}
```

参数 old_name 指向设备文件的路径名，而 new_name 指向安装点的路径名。读了 do_mount()的主流代码以后再来看这一段代码，可能会觉得比想像中的简单。实际上也确实是这样，原因在于通过回接设备安装之前事先要通过对回接设备的 ioctl()操作（具体的命令为 LOOP_SET_FD）在回接设备与目标设备（或格式化成文件系统的普通文件）之间建立起联系，有些准备工作已经在那时候做好了（有兴趣的读者可阅读 drivers/block/loop.c 中的 loop_set_fd()以及有关的代码）。

这里值得注意的是对 add_vfsmt()的调用参数，我们不妨把这里使用的调用参数与 do_mount()中使用的参数作一比较。对 add_vfsmt()的第一个调用参数是一个 nameidata 结构指针，代表着对待安装设备，在这里是回接设备（如/dev/loop0）节点的搜索结果，其中的一个 nameidata 成分指向该节点的 dentry 结构。第二个参数则是指向安装点的 dentry 结构的指针，这一点在两种情况下都一样。从概念上说，所谓“安装”正是要在安装点和待安装点这两个 dentry 结构之间架设起桥梁，从而建立起二者在逻辑上的等价性。这样一来，在 path_walk()中一旦到达安装的 dentry 结构就会通过 follow_down()进入回接设备的 dentry，就好像进入某个块设备根目录的 dentry 结构一样。所不同的是：在一般情况下进入的是块设备根目录的 dentry 结构，它的 inode 结构是通过该设备的驱动程序从设备上读入其根目录的索引节点而建立起来的。而如果进入的是回接设备的 dentry 结构呢？当内核企图通过回接设备的驱动程序从该虚拟的“块设备”上读入记录块时，却由回接设备的驱动程序“偷梁换柱”转到了原先设置好的目标设备，读入记录块的工作也“转包”给了目标设备的驱动程序。而回接设备本身的驱动程序，则变成了某种中介机构，并且从而可以在中间对过往的所有记录块进行加密、解密。所以，从“安装”本身的角度看，回接设备的安装确实很简单，但是通向目录设备的桥梁实际上是由回接设备的驱动程序通过一些内部的数据结构建立起来的。如果说由普通的块设备安装所建立的是直接的桥梁，那么由回接设备安装所建立的则是间接的桥梁，就好像在河中心有个小岛。从概念上说，这与在用户进程层

面上的输入 / 输出重定向以及通过管道实现的中间过滤进程是一致的。

看完了文件系统的安装，再来看文件系统的拆卸，这是由 sys_umount()完成的，其代码在 fs/super.c 中：

==================== fs/super.c 1109 1153 ====================

```
/*
 * Now umount can handle mount points as well as block devices.
 * This is important for filesystems which use unnamed block devices.
 *
 * We now support a flag for forced unmount like the other 'big iron'
 * unixes. Our API is identical to OSF/1 to avoid making a mess of AMD
 */

asmlinkage long sys_umount(char * name, int flags)
{
	struct nameidata nd;
	char *kname;
	int retval;

	lock_kernel();
	kname = getname(name);
	retval = PTR_ERR(kname);
	if (IS_ERR(kname))
		goto out;
	retval = 0;
	if (path_init(kname, LOOKUP_POSITIVE|LOOKUP_FOLLOW, &nd))
		retval = path_walk(kname, &nd);
	putname(kname);
	if (retval)
		goto out;
	retval = -EINVAL;
	if (nd.dentry != nd.mnt->mnt_root)
		goto dput_and_out;

	retval = -EPERM;
	if (!capable(CAP_SYS_ADMIN) && current->uid!=nd.mnt->mnt_owner)
		goto dput_and_out;

	dput(nd.dentry);
	/* puts nd.mnt */
	down(&mount_sem);
	retval = do_umount(nd.mnt, 0, flags);
	up(&mount_sem);
	goto out;
dput_and_out:
	path_release(&nd);
```

```
out:
    unlock_kernel();
    return retval;
}
```

由于 path_init()的调用参数中的 LOOKUP_FOLLOW 标志位为 1，不论给定的是安装点的路径名或是设备文件的路径名，path_walk()的结果都是一样的，nd.dentry 总是指向设备文件上根目录的 dentry 结构，而 nd.mnt 总是指向用来将该设备安装到安装点上的 vfsmount 数据结构。在安装设备的时候，总是将设备上的根目录作为该设备的代表安装到另一个设备上的某个节点上，所以如果 nd_dentry 不等于 nd.mnt->mnt_root 就说明有了严重的错误。通过了这一层检验，先把这个 dentry 结构释放，因为我们不再需要使用这个数据结构了。注意，这里的 nameidata 数据结构 nd 是局部量，所以并不需要释放其空间。至于 nd.mnt 所指向的 vfsmount 结构则还需要在 do_umount()中使用，所以释放这个数据结构的责任就转给了 do_umount()。完成文件系统拆卸操作的主体 do_umount()也在 fs/super.c 中：

==================== fs/super.c 1013 1055 ====================
[sys_umount()>do_umount()]

```
static int do_umount(struct vfsmount *mnt, int umount_root, int flags)
{
    struct super_block * sb = mnt->mnt_sb;

    /*
     * No sense to grab the lock for this test, but test itself looks
     * somewhat bogus. Suggestions for better replacement?
     * Ho-hum... In principle, we might treat that as umount + switch
     * to rootfs. GC would eventually take care of the old vfsmount.
     * The problem being: we have to implement rootfs and GC for that ;-)
     * Actually it makes sense, especially if rootfs would contain a
     * /reboot - static binary that would close all descriptors and
     * call reboot(9). Then init(8) could umount root and exec /reboot.
     */
    if (mnt == current->fs->rootmnt && !umount_root) {
        int retval = 0;
        /*
         * Special case for "unmounting" root ...
         * we just try to remount it readonly.
         */
        mntput(mnt);
        if (!(sb->s_flags & MS_RDONLY))
            retval = do_remount_sb(sb, MS_RDONLY, 0);
        return retval;
    }

    spin_lock(&dcache_lock);

    if (mnt->mnt_instances.next != mnt->mnt_instances.prev) {
        if (atomic_read(&mnt->mnt_count) > 2) {
            spin_unlock(&dcache_lock);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            mntput(mnt);
            return -EBUSY;
        }
        if (sb->s_type->fs_flags & FS_SINGLE)
            put_filesystem(sb->s_type);
        /* We hold two references, so mntput() is safe */
        mntput(mnt);
        remove_vfsmnt(mnt);
        return 0;
    }
    spin_unlock(&dcache_lock);

```

调用参数 umount_root 表示所要拆卸的是否根设备，我们在前面的代码中看到从 sys_umount()中调用时这个参数为 0，用户进程是不能通过 umount()直接拆卸根设备的。从用户进程通过 umount()系统调用拆卸根设备只意味着将它重安装成“只读”模式。

在 vfsmount 数据结构中也有个使用计数 mnt_count，在 add_vfsmnt()中设置为 1。从那以后，每当要使用这个数据结构时就通过 mntget()递增其使用计数，用完了就通过 mntput()递减其计数。例如，在函数 path_init()中就调用了 mntget()而在 path_release()中则调用了 mntput()；又如在 follow_up()和 follow_down()中都既调用了 mntget()又调用了 mntput()。所以，在 do_umount()中所处理的 vfsmount 结构中的使用计数应该是 2，如果大于这个数值就说明还有其它的操作过程还正在使用这个数据结构，因此不能完成拆卸而只能出错返回。当然，在出错返回之前也要通过 mntput()递减这个使用计数。

前面讲过，vfsmount 结构在安装文件系统时通过其队列头 mnt_instances 挂入一个 super_block 结构的 s_mounts 队列。通常一个块设备只安装一次，所以其 super_block 结构中的队列 s_mounts 只含有一个 vfsmount 结构，因此该 vfsmount 结构的队列头 mnt_instances 中的两个指针 next 和 prev 相等。但是，在有些情况下同一个设备是可以安装多次的，此时其 super_block 结构中的 s_mounts 队列含有多个 vfsmount 结构，而队列中的每个 vfsmount 结构的 mnt_instances 中的两个指针就不相等了。所以，此时代码中调用 remove_vfsmnt()所拆卸的并不是相应设备仅存的安装。这种情况下的拆卸比较简单，因为只是拆除该设备多次安装中的一次，而并非最终将设备拆下。下面给出函数 remove_vfsmnt()的代码（fs/super.c）：

==================== fs/super.c 408 428 ====================

[sys_umount()>do_umount()>remove_vfsmnt()]

```
/*
 * Called with spinlock held, releases it.
 */
static void remove_vfsmnt(struct vfsmount *mnt)
{
    /* First of all, remove it from all lists */
    list_del(&mnt->mnt_instances);
    list_del(&mnt->mnt_clash);
    list_del(&mnt->mnt_list);
    list_del(&mnt->mnt_child);
    spin_unlock(&dcache_lock);
    /* Now we can work safely */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (mnt->mnt_parent != mnt)
        mntput(mnt->mnt_parent);

    dput(mnt->mnt_mountpoint);
    dput(mnt->mnt_root);
    if (mnt->mnt_devname)
        kfree(mnt->mnt_devname);
    kfree(mnt);
}
```

对这些代码读者应该不会感到困难。函数 dput()递减一个 dentry 结构的使用计数，如果递减后达到了 0，就将此数据结构转移到 dentry_unused 队列中。

回到 do_umount()的代码中。相比之下，如果 vfsmount 数据结构代表着一个设备的惟一安装，那就比较复杂一点了。我们在这里并不关心磁盘空间配额的问题，所以跳过 DQUOT_OFF()和 acct_auto_close()直接往下读。

==================== fs/super.c 1056 1107 ====================

[sys_umount()>do_umount()]

```
1056      /*
1057       * Before checking whether the filesystem is still busy,
1058       * make sure the kernel doesn't hold any quota files open
1059       * on the device. If the umount fails, too bad -- there
1060       * are no quotas running any more. Just turn them on again.
1061       */
1062      DQUOT_OFF(sb);
1063      acct_auto_close(sb->s_dev);
1064
1065      /*
1066       * If we may have to abort operations to get out of this
1067       * mount, and they will themselves hold resources we must
1068       * allow the fs to do things. In the Unix tradition of
1069       * 'Gee thats tricky lets do it in userspace' the umount_begin
1070       * might fail to complete on the first run through as other tasks
1071       * must return, and the like. Thats for the mount program to worry
1072       * about for the moment.
1073       */
1074
1075      if( (flags&MNT_FORCE) && sb->s_op->umount_begin)
1076          sb->s_op->umount_begin(sb);
1077
1078      /*
1079       * Shrink dcache, then fsync. This guarantees that if the
1080       * filesystem is quiescent at this point, then (a) only the
1081       * root entry should be in use and (b) that root entry is
1082       * clean.
1083       */
1084      shrink_dcache_sb(sb);
1085      fsync_dev(sb->s_dev);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    if (sb->s_root->d_inode->i_state) {
        mntput(mnt);
        return -EBUSY;
    }

    /* Something might grab it again - redo checks */

    spin_lock(&dcache_lock);
    if (atomic_read(&mnt->mnt_count) > 2) {
        spin_unlock(&dcache_lock);
        mntput(mnt);
        return -EBUSY;
    }

    /* OK, that's the point of no return */
    mntput(mnt);
    remove_vfsmnt(mnt);

    kill_super(sb, umount_root);
    return 0;
}
```

有些设备要求在拆卸时先调用一个函数处理拆卸的开始，这种设备通过其 super_operations 函数跳转表内的函数指针 umount_begin 提供相应的函数。

把一个设备最终从文件系统中拆卸下来，这意味着从此以后这个子系统中的所有节点都不再是可访问的了。以前讲过，每当某个过程开始使用一个节点的 dentry 结构时都要通过 degt()递增其使用计数，如果内存中尚无此节点的 dentry 结构存在就要为之建立并将其使用计数设成 1。与其相对应，每当结束使用一个 dentry 结构时就要通过 dput()递减其使用计数，如果达到了 0 就要将这个数据结构转移到 dentry_unused 队列中。之所以不马上将不在使用中的 dentry 结构释放，而将它们留在这个队列中是为了提供一些缓冲，因为说不定很快就又要用了。可是现在既然要最终卸下一个设备，则属于这个设备的所有 dentry 结构再没有保留的必要。所以，此时要扫描 dentry_unused 队列，把所有属于这个队列的 dentry 结构都释放掉，这就是 shrink_dcache_sb()要做的事情。其代码在 fs/dcache.c 中：

==================== fs/dcache.c 378 428 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()]

```
/**
 * shrink_dcache_sb - shrink dcache for a superblock
 * @sb: superblock
 *
 * Shrink the dcache for the specified super block. This
 * is used to free the dcache before unmounting a file
 * system
 */

void shrink_dcache_sb(struct super_block * sb)
{
```

```
	struct list_head *tmp, *next;
	struct dentry *dentry;

	/*
	 * Pass one ... move the dentries for the specified
	 * superblock to the most recent end of the unused list.
	 */
	spin_lock(&dcache_lock);
	next = dentry_unused.next;
	while (next != &dentry_unused) {
		tmp = next;
		next = tmp->next;
		dentry = list_entry(tmp, struct dentry, d_lru);
		if (dentry->d_sb != sb)
			continue;
		list_del(tmp);
		list_add(tmp, &dentry_unused);
	}

	/*
	 * Pass two ... free the dentries for this superblock.
	 */
repeat:
	next = dentry_unused.next;
	while (next != &dentry_unused) {
		tmp = next;
		next = tmp->next;
		dentry = list_entry(tmp, struct dentry, d_lru);
		if (dentry->d_sb != sb)
			continue;
		if (atomic_read(&dentry->d_count))
			continue;
		dentry_stat.nr_unused--;
		list_del(tmp);
		INIT_LIST_HEAD(tmp);
		prune_one_dentry(dentry);
		goto repeat;
	}
	spin_unlock(&dcache_lock);
}
```

这段代码的逻辑比较简单，具体释放一个 dentry 结构的操作是由 prune_one_dentry()完成的，其代码在同一文件（dcache.c）中：

==================== fs/dcache.c 298 316 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()>prune_one_dentry()]

```
/*
 * Throw away a dentry - free the inode, dput the parent.
 * This requires that the LRU list has already been
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * removed.
 * Called with dcache_lock, drops it and then regains.
 */
static inline void prune_one_dentry(struct dentry * dentry)
{
	struct dentry * parent;

	list_del_init(&dentry->d_hash);
	list_del(&dentry->d_child);
	dentry_iput(dentry);
	parent = dentry->d_parent;
	d_free(dentry);
	if (parent != dentry)
		dput(parent);
	spin_lock(&dcache_lock);
}
```

再回到 do_umount()的代码中，下一件事是 fsync_dev()。

为了提高效率，块设备的输入 / 输出一般都是有缓冲的，无论是对超级块的改变还是对某个索引节点的改变，或者对某个数据块的改变，都只是对它们在内存中映象的改变，而不一定马上就写回设备上，现在设备要卸下来了，当然要先把已经改变了，但是尚未写回设备的内容写回去。这称为“同步”，是由 fsync_dev()完成的，其代码在 fs/buffer.c 中：

==================== fs/buffer.c 304 315 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()>fsync_dev()]

```
int fsync_dev(kdev_t dev)
{
	sync_buffers(dev, 0);

	lock_kernel();
	sync_supers(dev);
	sync_inodes(dev);
	DQUOT_SYNC(dev);
	unlock_kernel();

	return sync_buffers(dev, 1);
}
```

先看超级块的同步，函数 sync_supers()的代码在 fs/super.c 中：

==================== fs/super.c 605 629 ====================

```
/*
 * Note: check the dirty flag before waiting, so we don't
 * hold up the sync while mounting a device. (The newly
 * mounted device won't need syncing.)
 */
void sync_supers(kdev_t dev)
{
	struct super_block * sb;
```

```

    for (sb = sb_entry(super_blocks.next);
         sb != sb_entry(&super_blocks);
         sb = sb_entry(sb->s_list.next)) {
         if (!sb->s_dev)
              continue;
         if (dev && sb->s_dev != dev)
              continue;
         if (!sb->s_dirt)
              continue;
         lock_super(sb);
         if (sb->s_dev && sb->s_dirt && (!dev || dev == sb->s_dev))
              if (sb->s_op && sb->s_op->write_super)
                   sb->s_op->write_super(sb);
         unlock_super(sb);
    }
}
```

每当改变一个 super_block 结构的内容时都要将结构中的 s_dirt 标志设为 1，表示这个结构的内容已经“脏”了，也就是与设备上的超级块不一致了；而在将超级块写回设备时则将这个标志清 0。所以，如果一个 super_block 结构的 s_dirt 标志非 0 就表示应该加以同步。不过，如前所述，有些设备，主要是一些虚拟设备，本来就没有什么“超级块”或者类似的东西，所以还要看 super_block 结构中的指针 s_op 是否指向一个 super_operations 结构，以及这个结构中是否为 write_super 操作提供了一个函数。就 Ext2 文件系统而言，这个函数是 ext2_write_super()。读者可以在下面看了 ext2_read_super()以后自己阅读 ext2_write_super()。

再来看索引节点的同步，函数 sync_inodes()的代码在 fs/inode.c 中：

==================== fs/inode.c 237 265 ====================
[sys_umount()>do_umount()>shrink_dcache_sb()>fsync_dev()>sync_inodes()]

```
/**
 *  sync_inodes
 *  @dev: device to sync the inodes from.
 *
 *  sync_inodes goes through the super block's dirty list,
 *  writes them out, and puts them back on the normal list.
 */

void sync_inodes(kdev_t dev)
{
    struct super_block * sb = sb_entry(super_blocks.next);

    /*
     * Search the super_blocks array for the device(s) to sync.
     */
    spin_lock(&inode_lock);
    for (; sb != sb_entry(&super_blocks); sb = sb_entry(sb->s_list.next)) {
         if (!sb->s_dev)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                continue;
            if (dev && sb->s_dev != dev)
                continue;

            sync_list(&sb->s_dirty);

            if (dev)
                break;
        }
        spin_unlock(&inode_lock);
}
```

在 super_block 结构中还有个队列 s_dirty，凡是已经改变了的 inode 结构就通过它的 i_list 队列头挂入其所属 super_block 结构的 s_dirty 队列。所以，要同步的是整个队列，这是由 sync_list 完成的，有关的代码在同一文件（fs/inode.c）中：

==================== fs/inode.c 229 235 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()>fsync_dev()>sync_inodes()>sync_list()]

```
static inline void sync_list(struct list_head *head)
{
        struct list_head * tmp;

        while ((tmp = head->prev) != head)
            sync_one(list_entry(tmp, struct inode, i_list), 0);
}
```

==================== fs/inode.c 194 227 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()>fsync_dev()>sync_inodes()>sync_list()>sync_one()]

```
static inline void sync_one(struct inode *inode, int sync)
{
        if (inode->i_state & I_LOCK) {
            __iget(inode);
            spin_unlock(&inode_lock);
            __wait_on_inode(inode);
            iput(inode);
            spin_lock(&inode_lock);
        } else {
            unsigned dirty;

            list_del(&inode->i_list);
            list_add(&inode->i_list, atomic_read(&inode->i_count)
                                    ? &inode_in_use
                                    : &inode_unused);
            /* Set I_LOCK, reset I_DIRTY */
            dirty = inode->i_state & I_DIRTY;
            inode->i_state |= I_LOCK;
            inode->i_state &= ~I_DIRTY;
            spin_unlock(&inode_lock);

```

```
        filemap_fdatasync(inode->i_mapping);

        /* Don't write the inode if only I_DIRTY_PAGES was set */
        if (dirty & (I_DIRTY_SYNC | I_DIRTY_DATASYNC))
            write_inode(inode, sync);

        filemap_fdatawait(inode->i_mapping);

        spin_lock(&inode_lock);
        inode->i_state &= ~I_LOCK;
        wake_up(&inode->i_wait);
    }
}
```

==================== fs/inode.c 174 178 ====================

[sys_umount()>do_umount()>shrink_dcache_sb()>fsync_dev()>sync_inodes()>sync_list()>sync_one()>write_inode()]

```
static inline void write_inode(struct inode *inode, int sync)
{
    if (inode->i_sb && inode->i_sb->s_op && inode->i_sb->s_op->write_inode)
        inode->i_sb->s_op->write_inode(inode, sync);
}
```

我们把这些代码留给读者。Ext2 文件系统的 write_inode 操作为 ext2_write_inode()，由于我们在前面读过 ext2_read_inode()的代码，这里就不深入进去了。

由于我们对磁盘空间配额不感兴趣，剩下的只是数据块的同步了，那就是 sync_buffers()，我们将在“文件的读与写”一节中读这个函数的代码。

经过这些代码的阅读，读者对 super_block 数据结构想必已经有了个大致的印象，现在来看它的定义应该容易理解了。这是在 includc/linux/fs.h 中定义的：

==================== include/linux/fs.h 665 725 ====================

```
struct super_block {
    struct list_head    s_list;      /* Keep this first */
    kdev_t              s_dev;
    unsigned long       s_blocksize;
    unsigned char       s_blocksize_bits;
    unsigned char       s_lock;
    unsigned char       s_dirt;
    struct file_system_type *s_type;
    struct super_operations *s_op;
    struct dquot_operations *dq_op;
    unsigned long       s_flags;
    unsigned long       s_magic;
    struct dentry       *s_root;
    wait_queue_head_t   s_wait;

    struct list_head    s_dirty; /* dirty inodes */
    struct list_head    s_files;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	struct block_device	*s_bdev;
	struct list_head	s_mounts;/* vfsmount(s) of this one */
	struct quota_mount_options s_dquot;	/* Diskquota specific options */

	union {
		struct minix_sb_info	minix_sb;
		struct ext2_sb_info	ext2_sb;
		struct hpfs_sb_info	hpfs_sb;
		struct ntfs_sb_info	ntfs_sb;
		struct msdos_sb_info	msdos_sb;
		struct isofs_sb_info	isofs_sb;
		struct nfs_sb_info	nfs_sb;
		struct sysv_sb_info	sysv_sb;
		struct affs_sb_info	affs_sb;
		struct ufs_sb_info	ufs_sb;
		struct efs_sb_info	efs_sb;
		struct shmem_sb_info	shmem_sb;
		struct romfs_sb_info	romfs_sb;
		struct smb_sb_info	smbfs_sb;
		struct hfs_sb_info	hfs_sb;
		struct adfs_sb_info	adfs_sb;
		struct qnx4_sb_info	qnx4_sb;
		struct bfs_sb_info	bfs_sb;
		struct udf_sb_info	udf_sb;
		struct ncp_sb_info	ncpfs_sb;
		struct usbdev_sb_info	usbdevfs_sb;
		void			*generic_sbp;
	} u;
	/*
	 * The next field is for VFS *only*. No filesystems have any business
	 * even looking at it. You had been warned.
	 */
	struct semaphore s_vfs_rename_sem;	/* Kludge */

	/* The next field is used by knfsd when converting a (inode number based)
	 * file handle into a dentry. As it builds a path in the dcache tree from
	 * the bottom up, there may for a time be a subpath of dentrys which is not
	 * connected to the main tree.  This semaphore ensure that there is only ever
	 * one such free path per filesystem.  Note that unconnected files (or other
	 * non-directories) are allowed, but not unconnected diretories.
	 */
	struct semaphore s_nfsd_free_path_sem;
};
```

对于 Ext2 文件系统，将 super_block 结构中的 union 解释为一个 ext2_sb_info 结构，这是在 includ/linux/ext2_fs_sb.h 中定义的：

==================== include/linux/ext2_fs_sb.h 27 59 ====================

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * second extended-fs super-block data in memory
 */
struct ext2_sb_info {
     unsigned long s_frag_size;  /* Size of a fragment in bytes */
     unsigned long s_frags_per_block;/* Number of fragments per block */
     unsigned long s_inodes_per_block;/* Number of inodes per block */
     unsigned long s_frags_per_group;/* Number of fragments in a group */
     unsigned long s_blocks_per_group;/* Number of blocks in a group */
     unsigned long s_inodes_per_group;/* Number of inodes in a group */
     unsigned long s_itb_per_group;   /* Number of inode table blocks per group */
     unsigned long s_gdb_count;  /* Number of group descriptor blocks */
     unsigned long s_desc_per_block;  /* Number of group descriptors per block */
     unsigned long s_groups_count;    /* Number of groups in the fs */
     struct buffer_head * s_sbh; /* Buffer containing the super block */
     struct ext2_super_block * s_es;  /* Pointer to the super block in the buffer */
     struct buffer_head ** s_group_desc;
     unsigned short s_loaded_inode_bitmaps;
     unsigned short s_loaded_block_bitmaps;
     unsigned long s_inode_bitmap_number[EXT2_MAX_GROUP_LOADED];
     struct buffer_head * s_inode_bitmap[EXT2_MAX_GROUP_LOADED];
     unsigned long s_block_bitmap_number[EXT2_MAX_GROUP_LOADED];
     struct buffer_head * s_block_bitmap[EXT2_MAX_GROUP_LOADED];
     unsigned long  s_mount_opt;
     uid_t s_resuid;
     gid_t s_resgid;
     unsigned short s_mount_state;
     unsigned short s_pad;
     int s_addr_per_block_bits;
     int s_desc_per_block_bits;
     int s_inode_size;
     int s_first_ino;
};
```

如前所述，super_block 是内存中的数据结构，其内容通常（但并不总是）来自具体设备上特定文件系统的超级块。就 Ext2 文件系统而言，设备上的超级块为 ext2_super_block 结构，定义于 include/linux/ext2_fs.h 中：

==================== include/linux/ext2_fs.h 336 396 ====================

```
/*
 * Structure of the super block
 */
struct ext2_super_block {
     __u32     s_inodes_count;        /* Inodes count */
     __u32     s_blocks_count;        /* Blocks count */
     __u32     s_r_blocks_count;  /* Reserved blocks count */
     __u32     s_free_blocks_count;   /* Free blocks count */
     __u32     s_free_inodes_count;   /* Free inodes count */
     __u32     s_first_data_block;    /* First Data Block */
     __u32     s_log_block_size;  /* Block size */
```

```
	__s32	s_log_frag_size;	/* Fragment size */
	__u32	s_blocks_per_group;	/* # Blocks per group */
	__u32	s_frags_per_group;	/* # Fragments per group */
	__u32	s_inodes_per_group;	/* # Inodes per group */
	__u32	s_mtime;		/* Mount time */
	__u32	s_wtime;		/* Write time */
	__u16	s_mnt_count;		/* Mount count */
	__s16	s_max_mnt_count;	/* Maximal mount count */
	__u16	s_magic;		/* Magic signature */
	__u16	s_state;		/* File system state */
	__u16	s_errors;		/* Behaviour when detecting errors */
	__u16	s_minor_rev_level;	/* minor revision level */
	__u32	s_lastcheck;		/* time of last check */
	__u32	s_checkinterval;	/* max. time between checks */
	__u32	s_creator_os;		/* OS */
	__u32	s_rev_level;		/* Revision level */
	__u16	s_def_resuid;		/* Default uid for reserved blocks */
	__u16	s_def_resgid;		/* Default gid for reserved blocks */
	/*
	 * These fields are for EXT2_DYNAMIC_REV superblocks only.
	 *
	 * Note: the difference between the compatible feature set and
	 * the incompatible feature set is that if there is a bit set
	 * in the incompatible feature set that the kernel doesn't
	 * know about, it should refuse to mount the filesystem.
	 *
	 * e2fsck's requirements are more strict; if it doesn't know
	 * about a feature in either the compatible or incompatible
	 * feature set, it must abort and not try to meddle with
	 * things it doesn't understand...
	 */
	__u32	s_first_ino;		/* First non-reserved inode */
	__u16   s_inode_size;		/* size of inode structure */
	__u16	s_block_group_nr;	/* block group # of this superblock */
	__u32	s_feature_compat;	/* compatible feature set */
	__u32	s_feature_incompat;	/* incompatible feature set */
	__u32	s_feature_ro_compat;	/* readonly-compatible feature set */
	__u8	s_uuid[16];		/* 128-bit uuid for volume */
	char	s_volume_name[16];	/* volume name */
	char	s_last_mounted[64];	/* directory where last mounted */
	__u32	s_algorithm_usage_bitmap; /* For compression */
	/*
	 * Performance hints.  Directory preallocation should only
	 * happen if the EXT2_COMPAT_PREALLOC flag is on.
	 */
	__u8	s_prealloc_blocks;	/* Nr of blocks to try to preallocate*/
	__u8	s_prealloc_dir_blocks;	/* Nr to preallocate for dirs */
	__u16	s_padding1;
	__u32	s_reserved[204];	/* Padding to the end of the block */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
};
```

这个数据结构的定义与 Ext2 文件系统的格式密切相关，下面还要详述。建议读者将这几个数据结构的内容与下面 ext2_read_super()的代码相互参照印证，再回顾一下以前读过的代码，以求真正的理解。

最后，我们来看 ext2_read_super()的代码。这个函数在 fs/ext2/super.c 中，由于比较长，我们分段阅读。

==================== fs/ext2/super.c 384 434 ====================

[sys_mount()>do_mount()>get_sb_bdev()>read_super()>ext2_read_super()]

```
struct super_block * ext2_read_super (struct super_block * sb, void * data,
                        int silent)
{
     struct buffer_head * bh;
     struct ext2_super_block * es;
     unsigned long sb_block = 1;
     unsigned short resuid = EXT2_DEF_RESUID;
     unsigned short resgid = EXT2_DEF_RESGID;
     unsigned long logic_sb_block = 1;
     unsigned long offset = 0;
     kdev_t dev = sb->s_dev;
     int blocksize = BLOCK_SIZE;
     int hblock;
     int db_count;
     int i, j;

     /*
      * See what the current blocksize for the device is, and
      * use that as the blocksize.  Otherwise (or if the blocksize
      * is smaller than the default) use the default.
      * This is important for devices that have a hardware
      * sectorsize that is larger than the default.
      */
     blocksize = get_hardblocksize(dev);
     if( blocksize == 0 || blocksize < BLOCK_SIZE )
       {
         blocksize = BLOCK_SIZE;
       }

     sb->u.ext2_sb.s_mount_opt = 0;
     if (!parse_options ((char *) data, &sb_block, &resuid, &resgid,
         &sb->u.ext2_sb.s_mount_opt)) {
           return NULL;
     }

     set_blocksize (dev, blocksize);

     /*
      * If the superblock doesn't start on a sector boundary,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 * calculate the offset.  FIXME(eric) this doesn't make sense
	 * that we would have to do this.
	 */
	if (blocksize != BLOCK_SIZE) {
		logic_sb_block = (sb_block*BLOCK_SIZE) / blocksize;
		offset = (sb_block*BLOCK_SIZE) % blocksize;
	}

	if (!(bh = bread (dev, logic_sb_block, blocksize))) {
		printk ("EXT2-fs: unable to read superblock\n");
		return NULL;
	}
```

参数 sb 是指向 super_block 数据结构的指针，在调用这个函数之前对该结构已经作了一些初始化，例如其 s_dev 字段已经持有具体设备的设备号。但是，结构中的大部分内容都还没有设置，而这里要做的正是从设备上读入超级块并根据其内容设置这个 super_block 数据结构。另一个指针 data 的使用，则因文件系统而异，对于 Ext2 文件系统它是指向一个表示安装可选项的字符串。至于参数 silent，则表示在读超级块的过程中是否详细地报告出错信息。

首先是确定设备上记录块的大小。Ext2 文件系统的记录块大小一般是 1K 字节，但是为提高读写效率也可以采用 2K 字节或 4K 字节。变量 blocksize 先设置成常数 BLOCK_SIZE，即 1K 字节。但是，内核中有一个以主设备号为下标的指针数组 hardsect_size[]。如果这个数组中相应的元素指向另一个以次设备号为下标的整数数组，其中提供了该设备的记录块大小，并且这个数值大于 BLOCK_SIZE，则以此为准。这样，如果某种设备上的记录块大于 BLOCK_SIZE，便只要在系统初始化时设置这个数组中的相应元素就可以了。不过，从 hardsect_size[]中读时应通过为此而设的函数 get_handblocksize()进行。此外，在确定了某项设备的记录块大小以后要通过 set_blocksize()将确定了的记录块大小写回到这个数组中去，这样即使开始时数组是空的也会慢慢地得到设置。这些操作的逻辑比较简单，我们就不深入阅读这两个函数的代码了。值得注意的是 BLOCK_SIZE 实际上是记录块大小的最小值。

另一个函数 parse_optioms()是用来分析可选项字符串并根据其内容设置一些变量。每种文件系统都有它自己的 parse_options()，所以这些函数都是静态（static）函数，其作用域只是同一文件，如 Ext2 的 parse_options()就在 fs/ext2/super.c 中。函数 parse_options()通常都是既简单又冗长，所以我们不在这里列出其代码了。

超级块通常是设备上的 1 号记录块（即第 2 个记录块），所以变量 sb_block_size 设置为 1，在记录块大小为 BLOCK_SIZE 的设备上其逻辑块号 logic_sb_block 也是 1。但在记录块大于 BLOCK_SIZE 的设备上，由于超级块的大小仍为 BLOCK_SIZE，就要通过计算来确定其所在的记录块，以及在块内的位移。此时虽然仍称为超级"块"，但实际上只是记录块中的一部分了。确定了这两个参数以后，就可以通过 bread()将超级块所在的记录块读入内存了。函数 bread()属于设备驱动的范畴，读者可参阅下一章中的有关内容。我们继续往下看（fs/ext2/super.c）：

==================== fs/ext2/super.c 435 509 ====================

[sys_mount()>do_mount()>get_sb_bdev()>read_super()>ext2_read_super()]

```
	/*
	 * Note: s_es must be initialized s_es as soon as possible because
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 * some ext2 macro-instructions depend on its value
	 */
	es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
	sb->u.ext2_sb.s_es = es;
	sb->s_magic = le16_to_cpu(es->s_magic);
	if (sb->s_magic != EXT2_SUPER_MAGIC) {
		if (!silent)
			printk ("VFS: Can't find an ext2 filesystem on dev "
				"%s.\n", bdevname(dev));
	failed_mount:
		if (bh)
			brelse(bh);
		return NULL;
	}
	if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV &&
	    (EXT2_HAS_COMPAT_FEATURE(sb, ~0U) ||
	     EXT2_HAS_RO_COMPAT_FEATURE(sb, ~0U) ||
	     EXT2_HAS_INCOMPAT_FEATURE(sb, ~0U)))
		printk("EXT2-fs warning: feature flags set on rev 0 fs, "
		       "running e2fsck is recommended\n");
	/*
	 * Check feature flags regardless of the revision level, since we
	 * previously didn't change the revision level when setting the flags,
	 * so there is a chance incompat flags are set on a rev 0 filesystem.
	 */
	if ((i = EXT2_HAS_INCOMPAT_FEATURE(sb, ~EXT2_FEATURE_INCOMPAT_SUPP))) {
		printk("EXT2-fs: %s: couldn't mount because of "
		       "unsupported optional features (%x).\n",
		       bdevname(dev), i);
		goto failed_mount;
	}
	if (!(sb->s_flags & MS_RDONLY) &&
	    (i = EXT2_HAS_RO_COMPAT_FEATURE(sb, ~EXT2_FEATURE_RO_COMPAT_SUPP))){
		printk("EXT2-fs: %s: couldn't mount RDWR because of "
		       "unsupported optional features (%x).\n",
		       bdevname(dev), i);
		goto failed_mount;
	}
	sb->s_blocksize_bits =
		le32_to_cpu(EXT2_SB(sb)->s_es->s_log_block_size) + 10;
	sb->s_blocksize = 1 << sb->s_blocksize_bits;
	if (sb->s_blocksize != BLOCK_SIZE &&
	    (sb->s_blocksize == 1024 || sb->s_blocksize == 2048 ||
	     sb->s_blocksize == 4096)) {
		/*
		 * Make sure the blocksize for the filesystem is larger
		 * than the hardware sectorsize for the machine.
		 */
		hblock = get_hardblocksize(dev);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
486         if(   (hblock != 0)
487             && (sb->s_blocksize < hblock) )
488         {
489              printk("EXT2-fs: blocksize too small for device.\n");
490              goto failed_mount;
491         }
492
493         brelse (bh);
494         set_blocksize (dev, sb->s_blocksize);
495         logic_sb_block = (sb_block*BLOCK_SIZE) / sb->s_blocksize;
496         offset = (sb_block*BLOCK_SIZE) % sb->s_blocksize;
497         bh = bread (dev, logic_sb_block, sb->s_blocksize);
498         if(!bh) {
499              printk("EXT2-fs: Couldn't read superblock on "
500                     "2nd try.\n");
501              goto failed_mount;
502         }
503         es = (struct ext2_super_block *) (((char *)bh->b_data) + offset);
504         sb->u.ext2_sb.s_es = es;
505         if (es->s_magic != le16_to_cpu(EXT2_SUPER_MAGIC)) {
506              printk ("EXT2-fs: Magic mismatch, very weird !\n");
507              goto failed_mount;
508         }
509     }
```

函数 bread()返回一个 buffer_head 结构指针 bh，而 bh->data 就指向缓冲区，offset 则为超级块的起点在缓冲区中的位移。对于 Ext2 文件系统，从设备读入的超级块为一个 ext2_super_block 数据结构。从 439 行以后，指针 es 就指向这个数据结构。另一方面，Ext2 文件系统采用 " little ending"，所以一般而言对于超级块中的整数都要通过 le32_to_cpu()或 le16_to_cpu()变换成 CPU 所采用的制式。不过，由于 i386 结构本来就是采用 " little ending"，所以这些函数实际上不起作用。这样，结合前面三个数据结构的定义，这个函数中的大部分代码都是不难理解的，我们只择要讲几个问题。

从 475 行至 508 行是对记录块大小的修正。前面我们已经确定了设备上的记录块大小，但是那未必是来自设备本身的第一手信息。现在已经有了来自该设备的超级块，则超级块中提供的信息可能更为准确。如果发现超级块中提供的记录块大小与原来认为的不同（只能大，不能小），则一来要更正 hardsect_size[]数组中的内容，二来要把已经读入的 buffer_head 结构连同缓冲区释放（见 497 行）而根据新计算的参数再通过 bread()读入一次。

读者也许会感到奇怪，既然原来的参数不对，那怎么根据不正确的参数读入的超级块倒是对的呢？既然已经读入的超级块是对的，那何必又重新读一遍呢？原因就在于不管原来的参数是否正确，在 sb_block 等于 1 的前提下计算出来的 logic_sb_block 和 offset 只有两组结果。当 sb->s_blocksize 大于 BLOCK_SIZE 时，logic_sb_block 总是 0 而 offset 总是 BLOCK_SIZE，而与 sb->s_blocksize 的具体数值无关。当 sb->s_blocksize 等于 BLOCK_SIZE 时，则 logic_sb_block 为 1 而 offset 为 0。所以，只要在记录块大小等于 BLOCK_SIZE 时将超级块放在第二块（块号为 1），而在记录块大于 BLOCK_SIZE 时，则除将超级块放在第二块的开头处以外再在第一块中位移为 BLOCK_SIZE 处放上一个副本，就不会错了。这里重新读一遍，只不过是让缓冲区中含有整个记录块，而不只是超级块而已。同时，在 super_block

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构中也保留着两个指针，一个指向缓冲区中超级块的起点（见 440 行和 504 行），另一个则指向缓冲区本身（见 538 行）。

记录块的大小是个重要的参数。从读 / 写的效率考虑，记录块大一些较好，但是，记录块大了往往造成空间的浪费，因为记录块是设备上存储空间分配的单位。据统计，在 Unix（以及 Linux）环境下大多数文件都是比较小的，这样，浪费的百分之比就更大了。权衡之下，Ext2 选择 1K 字节为默认的记录块大小，但是也可以在格式化时给定更大的数值，这就是前面有关记录块大小的处理的来历。由于时间上和空间上的效率难于兼顾，有些文件系统进一步把记录块划分成若干“片断”（fragment），当需要的空间较小时就以“片断”为分配单位，Ext2 也准备采用这项技术，并且在数据结构等方面为此作好了准备（所以超级块中有“片断大小”等字段），但是从总体来说尚未实现，因此目前“片断大小”总是等于“记录块大小”。

超级块的内容反映了按特定格式建立在特定设备上的文件系统多方面的信息，主要是结构和管理两方面的信息。其中结构方面的信息是与具体文件系统的格式密切相关的，所以要了解 Ext2 文件系统的格式才能理解其超级块的内容。以前提到过，Ext2 文件系统的第一个记录块为引导块，第二个记录块为超级块，然后是索引节点区，接着是数据区。但是，那只是从概念上讲，是大大简化了的，实际上要复杂得多。现代的磁盘驱动器都是多片的，所以不同盘面上的相同磁道合在一起就形成了“柱面 ”（cylinder）的概念。从磁盘读出多个记录块时，如果是从同一柱面中读出就比较快，因为在这种情况下不需要移动磁头（实际上是磁头组）。互相连续的记录块实际上分布在同一柱面的各个盘面上，只有在一个柱面用满后方进入下一个柱面。所以，在许多文件系统中都把整个设备划分成若干“柱面组”，将反映着盘面存储空间的组织与管理的信息分散后就近存储于各个柱面组中。相比之下，早期的文件系统往往将这些信息集中存储在一起，使得磁头在文件访问时来回“疲于奔命”而降低了效率。

但是，柱面组的划分也带来了一些新的、附加的要求。首先是关于这些柱面组本身的结构信息，如此就要用一些记录块来保存所有的柱面组的描述，即所谓“组描述结构”（group descriptor）。另一方面，有些信息是对于整个设备的而不只是针对一个柱面组的，所以不能把它拆散，而只能重复地存储于每个柱面组中。从另一个角度来讲，将某些重要的信息重复存储于每个柱面组为这些信息提供了后备，从而增加了可靠性。对于文件系统来说，最重要的莫过于其超级块了，所以一些文件系统的设计要求设备上不管哪一个记录块、哪一个盘面、哪一个磁道坏了都仍然能恢复其超级块（通过运行 fsck）。Ext2 也采用了这样的结构，不过不称为“柱面组”而称为“记录块组”，并且将超级块和所有的块组描述结构重复存储于每个块组。此外，Ext2 通过“位图”来管理每个块组中的记录块和索引节点，所以在每个块组中有两个位图，一个用于记录块，一个用于索引节点。这样，Ext2 文件系统的格式就变成了如图 5.5 所示的形式。

图 5.5 中的组描述块为记录着全部组描述结构的记录块，具体的块数取决于设备的大小。记录块位图则是本块组的位图（每 1 位对应着块组中的一个记录块，1 表示已分配，0 表示空闲），占用的块数取决于块组的大小。当记录块大小为 1K 字节而块组的大小为 8192 时，该位图恰好占一个记录块。用于索引节点的记录块数量取决于文件系统的参数，而索引节点的位图则不会超出一个记录块。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

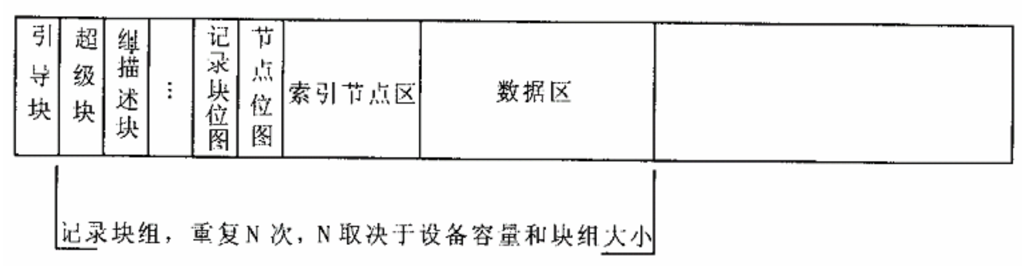


图 5.5　Ext2 文件系统格式示意图

当整个设备上只有一个块组时，就简化成了以前讲过的那种结构。Ext2 的块组描述结构定义于 include/linux/ext2_fs.h：

==================== include/linux/ext2_fs.h 145 158 ====================

```
/*
 * Structure of a blocks group descriptor
 */
struct ext2_group_desc
{
	__u32	bg_block_bitmap;		/* Blocks bitmap block */
	__u32	bg_inode_bitmap;		/* Inodes bitmap block */
	__u32	bg_inode_table;		/* Inodes table block */
	__u16	bg_free_blocks_count;	/* Free blocks count */
	__u16	bg_free_inodes_count;	/* Free inodes count */
	__u16	bg_used_dirs_count;	/* Directories count */
	__u16	bg_pad;
	__u32	bg_reserved[3];
};
```

超级块的内容可以用“/sbin/tune2fs -l”、“/sbin/dumpe2fs”等命令来显示。下面我们通过一个实例来看一个磁盘设备的组织，以便读者加深理解。这个设备就是笔者机器上的/dev/hda2。先用“tune2fs -l /dev/hda2”观察该设备上超级块的内容，我们关心的有下面这些：

```
Inode count:             386528
Block count:             1540097
Reserved block count:     77011
Free blocks:             221060
First block:             1
Block size:              1024
Blocks per group:         8192
Inodes per group:         2056
Inode blocks per group:   257
```

让我们看看这些数字是怎样互相联系的。

首先，这个设备上可用的记录块数量为 1540097，也就是大约 1.5G 字节（在格式化时可能会浪费少量记录块），而每个块组的大小为 8192，所以设备上共有 1540097 / 8192 = 188 个块组。但是 8192 × 188

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

= 1540096，这多余的一个记录块就是引导块；而第一个实际属于该文件系统的记录块的块号为 1。其次，每个块组含有 2056 个索引节点，所以总共有 2056 × 188 = 386528 个索引节点。由于记录块大小为 1K 字节，而索引节点的大小为 128 字节，所以每个记录块可容纳 8 个索引节点，这样，每个块组将 2056 / 8 = 257 个记录块用于索引节点。目前设备上尚有 221060 个空闲的记录块，但是其中 77011 个是保留的，保留的记录块通常占总容量的 5%左右，当某些记录块损坏时就用保留的记录块作为替换。

再通过 “ df”，命令来印证一下，显示的结果表明该设备共有 1490088 个 1K 字节的记录块，其中已用去 1269028 个，尚有 144049 个记录块。这些数字怎样与上面的数字相联系呢？这里的 1490088 就是设备上真正用于数据块的记录块数量，也就是每个块组有 1490088 / 188 = 7926 个数据块。我们知道每个块组还有 257 个记录块用于索引节点，这样一共是 8183 个，还有 9 个记录块干什么用了呢？请看：

1（超级块）+ 6（块组描述结构）+ 1（记录块位图）+ 1（索引节点位图）= 9

每个 Ext2 记录块描述结构的大小是 32 字节，每 1K 字节的记录块能容纳 32 个块描述结构，因此共需 188 / 32 = 6 个记录块（经过取整）用于块组描述结构。

再看可分配使用的记录块数量。总共 1490088 个记录块，已用去 1269028 个，应该还有 221060 个，怎么说只剩 144049 个了呢？这是因为有 77011 个记录块是保留的，而 221060 - 77011 = 144049。

最后还可以通过命令 “ dumpe2fs　/dev/hda2” 观察每个块组的详情。

现在读者对 Ext2 文件系统的结构已经有了个比较直观的了解，再回过去读 ext2_read_super()中余下的代码就容易一些了。

代码中用到的一些宏定义基本上是不言自明的，这里值得一提的是，在 ext2_sb_info 结构中的数组 s_inode_bitmap_number[]和 s_block_numer[]都是固定大小的，具体位图数组也是固定大小的，大小为 EXT2_MAX_GROUP_LOADED。常数 EXT2_MAX_GROUP_LOADED 在 include/linux/ext2_fs.h 中定义为 8，与块组的总数一比只占很小的比例，所以运行时并不是将所有块组的位图都装入到这些数组中，而是只装入其中很小一部分，根据具体运行的需要周转。这里只是把这些数组以及有关的变量都初始化成空白，也并没有为位图数组本身分配空间。

继续往下看 ext2_read_super()的代码：

==================== fs/ext2/super.c 510 642 ====================
[sys_mount()>do_mount()>get_sb_bdev()>read_super()>ext2_read_super()]

```
	if (le32_to_cpu(es->s_rev_level) == EXT2_GOOD_OLD_REV) {
		sb->u.ext2_sb.s_inode_size = EXT2_GOOD_OLD_INODE_SIZE;
		sb->u.ext2_sb.s_first_ino = EXT2_GOOD_OLD_FIRST_INO;
	} else {
		sb->u.ext2_sb.s_inode_size = le16_to_cpu(es->s_inode_size);
		sb->u.ext2_sb.s_first_ino = le32_to_cpu(es->s_first_ino);
		if (sb->u.ext2_sb.s_inode_size != EXT2_GOOD_OLD_INODE_SIZE) {
			printk ("EXT2-fs: unsupported inode size: %d\n",
				sb->u.ext2_sb.s_inode_size);
			goto failed_mount;
		}
	}
	sb->u.ext2_sb.s_frag_size = EXT2_MIN_FRAG_SIZE <<
				   le32_to_cpu(es->s_log_frag_size);
	if (sb->u.ext2_sb.s_frag_size)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        sb->u.ext2_sb.s_frags_per_block = sb->s_blocksize /
                          sb->u.ext2_sb.s_frag_size;
    else
        sb->s_magic = 0;
    sb->u.ext2_sb.s_blocks_per_group = le32_to_cpu(es->s_blocks_per_group);
    sb->u.ext2_sb.s_frags_per_group = le32_to_cpu(es->s_frags_per_group);
    sb->u.ext2_sb.s_inodes_per_group = le32_to_cpu(es->s_inodes_per_group);
    sb->u.ext2_sb.s_inodes_per_block = sb->s_blocksize /
                       EXT2_INODE_SIZE(sb);
    sb->u.ext2_sb.s_itb_per_group = sb->u.ext2_sb.s_inodes_per_group /
                      sb->u.ext2_sb.s_inodes_per_block;
    sb->u.ext2_sb.s_desc_per_block = sb->s_blocksize /
                     sizeof (struct ext2_group_desc);
    sb->u.ext2_sb.s_sbh = bh;
    if (resuid != EXT2_DEF_RESUID)
        sb->u.ext2_sb.s_resuid = resuid;
    else
        sb->u.ext2_sb.s_resuid = le16_to_cpu(es->s_def_resuid);
    if (resgid != EXT2_DEF_RESGID)
        sb->u.ext2_sb.s_resgid = resgid;
    else
        sb->u.ext2_sb.s_resgid = le16_to_cpu(es->s_def_resgid);
    sb->u.ext2_sb.s_mount_state = le16_to_cpu(es->s_state);
    sb->u.ext2_sb.s_addr_per_block_bits =
        log2 (EXT2_ADDR_PER_BLOCK(sb));
    sb->u.ext2_sb.s_desc_per_block_bits =
        log2 (EXT2_DESC_PER_BLOCK(sb));
    if (sb->s_magic != EXT2_SUPER_MAGIC) {
        if (!silent)
            printk ("VFS: Can't find an ext2 filesystem on dev "
                "%s.\n",
                bdevname(dev));
        goto failed_mount;
    }
    if (sb->s_blocksize != bh->b_size) {
        if (!silent)
            printk ("VFS: Unsupported blocksize on dev "
                "%s.\n", bdevname(dev));
        goto failed_mount;
    }

    if (sb->s_blocksize != sb->u.ext2_sb.s_frag_size) {
        printk ("EXT2-fs: fragsize %lu != blocksize %lu (not supported yet)\n",
            sb->u.ext2_sb.s_frag_size, sb->s_blocksize);
        goto failed_mount;
    }

    if (sb->u.ext2_sb.s_blocks_per_group > sb->s_blocksize * 8) {
        printk ("EXT2-fs: #blocks per group too big: %lu\n",
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                sb->u.ext2_sb.s_blocks_per_group);
        goto failed_mount;
    }
    if (sb->u.ext2_sb.s_frags_per_group > sb->s_blocksize * 8) {
        printk ("EXT2-fs: #fragments per group too big: %lu\n",
                sb->u.ext2_sb.s_frags_per_group);
        goto failed_mount;
    }
    if (sb->u.ext2_sb.s_inodes_per_group > sb->s_blocksize * 8) {
        printk ("EXT2-fs: #inodes per group too big: %lu\n",
                sb->u.ext2_sb.s_inodes_per_group);
        goto failed_mount;
    }

    sb->u.ext2_sb.s_groups_count = (le32_to_cpu(es->s_blocks_count) -
                                    le32_to_cpu(es->s_first_data_block) +
                                   EXT2_BLOCKS_PER_GROUP(sb) - 1) /
                                   EXT2_BLOCKS_PER_GROUP(sb);
    db_count = (sb->u.ext2_sb.s_groups_count + EXT2_DESC_PER_BLOCK(sb) - 1) /
           EXT2_DESC_PER_BLOCK(sb);
    sb->u.ext2_sb.s_group_desc = kmalloc (db_count * sizeof (struct buffer_head *), GFP_KERNEL);
    if (sb->u.ext2_sb.s_group_desc == NULL) {
        printk ("EXT2-fs: not enough memory\n");
        goto failed_mount;
    }
    for (i = 0; i < db_count; i++) {
        sb->u.ext2_sb.s_group_desc[i] = bread (dev, logic_sb_block + i + 1,
                                               sb->s_blocksize);
        if (!sb->u.ext2_sb.s_group_desc[i]) {
            for (j = 0; j < i; j++)
                brelse (sb->u.ext2_sb.s_group_desc[j]);
            kfree(sb->u.ext2_sb.s_group_desc);
            printk ("EXT2-fs: unable to read group descriptors\n");
            goto failed_mount;
        }
    }
    if (!ext2_check_descriptors (sb)) {
        for (j = 0; j < db_count; j++)
            brelse (sb->u.ext2_sb.s_group_desc[j]);
        kfree(sb->u.ext2_sb.s_group_desc);
        printk ("EXT2-fs: group descriptors corrupted !\n");
        goto failed_mount;
    }
    for (i = 0; i < EXT2_MAX_GROUP_LOADED; i++) {
        sb->u.ext2_sb.s_inode_bitmap_number[i] = 0;
        sb->u.ext2_sb.s_inode_bitmap[i] = NULL;
        sb->u.ext2_sb.s_block_bitmap_number[i] = 0;
        sb->u.ext2_sb.s_block_bitmap[i] = NULL;
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
623      sb->u.ext2_sb.s_loaded_inode_bitmaps = 0;
624      sb->u.ext2_sb.s_loaded_block_bitmaps = 0;
625      sb->u.ext2_sb.s_gdb_count = db_count;
626      /*
627       * set up enough so that it can read an inode
628       */
629      sb->s_op = &ext2_sops;
630      sb->s_root = d_alloc_root(iget(sb, EXT2_ROOT_INO));
631      if (!sb->s_root) {
632          for (i = 0; i < db_count; i++)
633              if (sb->u.ext2_sb.s_group_desc[i])
634                  brelse (sb->u.ext2_sb.s_group_desc[i]);
635          kfree(sb->u.ext2_sb.s_group_desc);
636          brelse (bh);
637          printk ("EXT2-fs: get root inode failed\n");
638          return NULL;
639      }
640      ext2_setup_super (sb, es, sb->s_flags & MS_RDONLY);
641      return sb;
642  }
```

这段代码虽然比较长，却并不复杂（对于读者已有的基础来说），所以我们基本上把它留给读者，而只注意其尾部。

超级块只是反映具体设备上文件系统的组织和管理的信息，而并不涉及该文件系统的内容，设备上“根目录”的索引节点才是打开这个文件系统的钥匙。文件系统中的每一个文件，包括目录，都有一个索引节点，其节点号必然存在于该文件所在的目录项之中，唯有根目录的索引节点号是固定的，那就是 EXT2_ROOT_INO，即 2 号索引节点。代码中的第 598 行先通过 iget()将这个索引节点读入内存并为之建立 inode 数据结构，再通过 d_alloc_root()在内存中为之建立起一个 dentry 数据结构，并使 super_block 结构中的指针 s_root 指向这个 dentry 结构。这样，通向这个文件系统的途径就可以建立起来了。函数 d_alloc_root()的代码见下。

==================== fs/dcache.c 672 694 ====================

[sys_mount()>do_mount()>get_sb_bdev()>read_super()>ext2_read_super()>d_alloc_root()]

```
672  /**
673   * d_alloc_root - allocate root dentry
674   * @root_inode: inode to allocate the root for
675   *
676   * Allocate a root ("/") dentry for the inode given. The inode is
677   * instantiated and returned. %NULL is returned if there is insufficient
678   * memory or the inode passed is %NULL.
679   */
680
681  struct dentry * d_alloc_root(struct inode * root_inode)
682  {
683      struct dentry *res = NULL;
684
685      if (root_inode) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
686          res = d_alloc(NULL, &(const struct qstr) { "/", 1, 0 });
687          if (res) {
688              res->d_sb = root_inode->i_sb;
689              res->d_parent = res;
690              d_instantiate(res, root_inode);
691          }
692      }
693      return res;
694  }
```

如前所述，根目录的索引节点号是固定的，它也不出现在哪个目录中。所以，根目录是无名的，而为根目录建立的 dentry 数据结构则都以“ /”为名。代码中的&(const struct qstr){"/", 1, 0}表示一个 qstr 结构指针，它所指向的 qstr 结构为{"/", 1, 0}，即节点名为“ /”，节点名长度为 1；数据结构的内容为常量，不允许改变。

最后（640 行）是调用 ext2_setup_super()设置一些与管理有关的信息，包括此次安装的时间以及递增安装计数，当安装计数达到某个最大值时，就应该对这个文件系统运行 e2fsck 加以检验了。另一方面，由于每安装一次，超级块的内容就一定有些变化（至少是安装计数），所以要将超级块的缓冲区标志记成“脏”。函数 ext2_setup_super()的代码在 fs/ext2/super.c 中，代码很简单，我们就不解释了，请读者自行阅读。

==================== fs/ext2/super.c 283 333 ====================

[sys_mount()>do_mount()>get_sb_bdev()>read_super()>ext2_read_super()>ext2_setup_super()]

```
283  static int ext2_setup_super (struct super_block * sb,
284                     struct ext2_super_block * es,
285                     int read_only)
286  {
287      int res = 0;
288      if (le32_to_cpu(es->s_rev_level) > EXT2_MAX_SUPP_REV) {
289          printk ("EXT2-fs warning: revision level too high, "
290              "forcing read-only mode\n");
291          res = MS_RDONLY;
292      }
293      if (read_only)
294          return res;
295      if (!(sb->u.ext2_sb.s_mount_state & EXT2_VALID_FS))
296          printk ("EXT2-fs warning: mounting unchecked fs, "
297              "running e2fsck is recommended\n");
298      else if ((sb->u.ext2_sb.s_mount_state & EXT2_ERROR_FS))
299          printk ("EXT2-fs warning: mounting fs with errors, "
300              "running e2fsck is recommended\n");
301      else if ((__s16) le16_to_cpu(es->s_max_mnt_count) >= 0 &&
302           le16_to_cpu(es->s_mnt_count) >=
303           (unsigned short) (__s16) le16_to_cpu(es->s_max_mnt_count))
304          printk ("EXT2-fs warning: maximal mount count reached, "
305              "running e2fsck is recommended\n");
306      else if (le32_to_cpu(es->s_checkinterval) &&
307          (le32_to_cpu(es->s_lastcheck) + le32_to_cpu(es->s_checkinterval) <= CURRENT_TIME))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            printk ("EXT2-fs warning: checktime reached, "
                "running e2fsck is recommended\n");
        es->s_state = cpu_to_le16(le16_to_cpu(es->s_state) & ~EXT2_VALID_FS);
        if (!(__s16) le16_to_cpu(es->s_max_mnt_count))
            es->s_max_mnt_count = (__s16) cpu_to_le16(EXT2_DFL_MAX_MNT_COUNT);
        es->s_mnt_count=cpu_to_le16(le16_to_cpu(es->s_mnt_count) + 1);
        es->s_mtime = cpu_to_le32(CURRENT_TIME);
        mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
        sb->s_dirt = 1;
        if (test_opt (sb, DEBUG))
            printk ("[EXT II FS %s, %s, bs=%lu, fs=%lu, gc=%lu, "
                "bpg=%lu, ipg=%lu, mo=%04lx]\n",
                EXT2FS_VERSION, EXT2FS_DATE, sb->s_blocksize,
                sb->u.ext2_sb.s_frag_size,
                sb->u.ext2_sb.s_groups_count,
                EXT2_BLOCKS_PER_GROUP(sb),
                EXT2_INODES_PER_GROUP(sb),
                sb->u.ext2_sb.s_mount_opt);
    #ifdef CONFIG_EXT2_CHECK
        if (test_opt (sb, CHECK)) {
            ext2_check_blocks_bitmap (sb);
            ext2_check_inodes_bitmap (sb);
        }
    #endif
        return res;
    }
```

至此，ext2_read_super()的工作就全部完成了，函数返回指向该 super_block 数据结构的指针。

5.5 文件的打开与关闭

用户进程在能够读 / 写一个文件之前必须要先“打开”这个文件。对文件的读 / 写从概念上说是一种进程与文件系统之间的一种“有连接”通信，所谓“打开文件”实质上就是在进程与文件之间建立起连接，而“打开文件号”就惟一地标识着这样一个连接。不过，严格意义上的“连接”意味着一个独立的“上下文”，如果一个进程与某个目标之间重复建立起多个连接，则每个连接都应该是互相独立的。在文件系统的处理中，每当一个进程重复打开同一个文件时就建立起一个由 file 数据结构代表的独立的上下文。通常，一个 file 数据结构，即一个读 / 写文件的上下文，都由一个“打开文件号 ”加以标识，但是通过系统调用 dup()或 dup2()却可以使同一个 file 结构对应到多个“打开文件号。”

打开文件的系统调用是 open()，在内核中通过 sys_open()实现，其代码在 fs/open.c 中：

==================== fs/open.c 743 771 ====================

```
asmlinkage long sys_open(const char * filename, int flags, int mode)
{
    char * tmp;
    int fd, error;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

#if BITS_PER_LONG != 32
     flags |= O_LARGEFILE;
#endif
     tmp = getname(filename);
     fd = PTR_ERR(tmp);
     if (!IS_ERR(tmp)) {
          fd = get_unused_fd();
          if (fd >= 0) {
               struct file *f = filp_open(tmp, flags, mode);
               error = PTR_ERR(f);
               if (IS_ERR(f))
                    goto out_error;
               fd_install(fd, f);
          }
out:
          putname(tmp);
     }
     return fd;

out_error:
     put_unused_fd(fd);
     fd = error;
     goto out;
}
```

调用参数 filename 实际上是文件的路径名（绝对路径名或相对路径名）；mode 表示打开的模式，如“只读”等等；而 flag 则包含了许多标志位，用以表示打开模式以外的一些属性和要求。函数通过 getname()从用户空间把文件的路径名拷贝到系统空间，并通过 get_unused_fd()从当前进程的“打开文件表”中找到一个空闲的表项，该表项的下标即为“打开文件号”。然后，根据文件名通过 file_open()找到或创建一个“连接”，或者说读／写该文件的上下文。文件读写的上下文是由 file 数据结构代表和描绘的，其定义见 include/linux/fs.h：

==================== include/linux/fs.h 498 516 ====================

```
struct file {
     struct list_head   f_list;
     struct dentry      *f_dentry;
     struct vfsmount        *f_vfsmnt;
     struct file_operations *f_op;
     atomic_t      f_count;
     unsigned int       f_flags;
     mode_t             f_mode;
     loff_t             f_pos;
     unsigned long      f_reada, f_ramax, f_raend, f_ralen, f_rawin;
     struct fown_struct f_owner;
     unsigned int       f_uid, f_gid;
     int           f_error;

```

```
	unsigned long		f_version;

	/* needed for tty driver, and maybe others */
	void			*private_data;
};
```

数据结构中不但有指向文件的 dentry 结构的指针 f_dentry，指向将文件所在设备安装在文件系统中的 vfsmnt 结构的指针，有共享计数 f_count，还有一个在文件中的当前读写位置 f_pos，这就是“上下文”。找到或者创建了代表着目标文件的 file 结构以后，就通过 fd_install()将指向这个结构的指针填入当前进程的打开文件表，即由其 task_sturct 结构中的指针 file 所指向的 files_struct 数组中并返回其数组中的下标。

函数 get_unused_fd()的代码也在 fs/open.c 中：

==================== fs/open.c 681 741 ====================

[sys_open()>get_unused_fd()]

```
/*
 * Find an empty file descriptor entry, and mark it busy.
 */
int get_unused_fd(void)
{
	struct files_struct * files = current->files;
	int fd, error;

	error = -EMFILE;
	write_lock(&files->file_lock);

repeat:
	fd = find_next_zero_bit(files->open_fds,
				files->max_fdset,
				files->next_fd);

	/*
	 * N.B. For clone tasks sharing a files structure, this test
	 * will limit the total number of files that can be opened.
	 */
	if (fd >= current->rlim[RLIMIT_NOFILE].rlim_cur)
		goto out;

	/* Do we need to expand the fdset array? */
	if (fd >= files->max_fdset) {
		error = expand_fdset(files, fd);
		if (!error) {
			error = -EMFILE;
			goto repeat;
		}
		goto out;
	}

```

```
        /*
         * Check whether we need to expand the fd array.
         */
        if (fd >= files->max_fds) {
                error = expand_fd_array(files, fd);
                if (!error) {
                        error = -EMFILE;
                        goto repeat;
                }
                goto out;
        }

        FD_SET(fd, files->open_fds);
        FD_CLR(fd, files->close_on_exec);
        files->next_fd = fd + 1;
#if 1
        /* Sanity check */
        if (files->fd[fd] != NULL) {
                printk("get_unused_fd: slot %d not NULL!\n", fd);
                files->fd[fd] = NULL;
        }
#endif
        error = fd;

out:
        write_unlock(&files->file_lock);
        return error;
}
```

进程的 task_struct 结构中有个指针 files，指向本进程的 files_struct 数据结构。与打开文件有关的信息都保存在这个数据结构中，其定义在 include/linux/sched.h 中：

==================== include/linux/sched.h 159 180 ====================

```
/*
 * The default fd array needs to be at least BITS_PER_LONG,
 * as this is the granularity returned by copy_fdset().
 */
#define NR_OPEN_DEFAULT BITS_PER_LONG

/*
 * Open file table structure
 */
struct files_struct {
        atomic_t count;
        rwlock_t file_lock;
        int max_fds;
        int max_fdset;
        int next_fd;
        struct file ** fd; /* current fd array */
```

```
    fd_set *close_on_exec;
    fd_set *open_fds;
    fd_set close_on_exec_init;
    fd_set open_fds_init;
    struct file * fd_array[NR_OPEN_DEFAULT];
};
```

这个数据结构中最主要的成分是一个 file 结构指针数组 fd_array[]，这个数组的大小是固定的，即 32，其下标即为“打开文件号”。另外，结构中还有个指针 fd，最初时指向 fd_array[]。结构中还有两个位图 close_on_exec_init 和 open_fds_init，这些位图大致对应着 file 结构指针数组的内容，但是比 fd_array[]的大小要大得多。同时，又有两个指针 close_on_exec 和 open_fds，最初时分别指向上述两个位图。每次打开文件分配一个打开文件号时就将由 open_fds 所指向位图中的相应位设成 1。此外，该数据结构中还有两个参数 max_fds 和 max_fdset，分别反映着当前 file 结构指针数组与位图的容量。一个进程可以有多少个已打开文件只取决于该进程的 task_struct 结构中关于可用资源的限制（见上面代码中的第 701 行）。在这个限制以内，如果超出了其 file 结构指针数组的容量就通过 expand_fd_array()扩充该数组的容量，并让指针 fd 指向新的数组；如果超出了位图的容量就通过 expand_fdset()扩充两个位图的容量，并使两个指针也分别指向新的位图。这样，就克服了早期 Unix 因只采用固定大小的 file 结构指针数组而使每个进程可以同时打开文件数量受到限制的缺陷。

打开文件时，更确切地说是分配空闲打开文件号时，通过宏操作 FD_SET()将 open_fds 所指向的位图中的相应位设成 1，表示这个打开文件号已不再空闲，这个位图代表着已经在使用中的打开文件号。同时，还通过 FD_CLR()将由指针 close_on_exec 所指向的位图中的相应位清 0，表示如果当前进程通过 exec()系统调用执行一个可执行程序的话无需将这个文件关闭。这个位图的内容可以通过 ioctl()系统调用来设置。

动态地调整可同时打开的文件数量对于现代、特别是面向对象的环境具有重要意义，因为在这些环境下常常要求同时打开数量众多（但是每个文件却很小）的文件。

显然，sys_open()的主体是 filp_open()，其代码也在 fs/open.c 中：

==================== fs/open.c 600 630 ====================

[sys_open()>filp_open()]

```
/*
 * Note that while the flag value (low two bits) for sys_open means:
 *    00 - read-only
 *    01 - write-only
 *    10 - read-write
 *    11 - special
 * it is changed into
 *    00 - no permissions needed
 *    01 - read-permission
 *    10 - write-permission
 *    11 - read-write
 * for the internal routines (ie open_namei()/follow_link() etc). 00 is
 * used by symlinks.
 */
struct file *filp_open(const char * filename, int flags, int mode)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
615 {
616      int namei_flags, error;
617      struct nameidata nd;
618
619      namei_flags = flags;
620      if ((namei_flags+1) & O_ACCMODE)
621           namei_flags++;
622      if (namei_flags & O_TRUNC)
623           namei_flags |= 2;
624
625      error = open_namei(filename, namei_flags, mode, &nd);
626      if (!error)
627           return dentry_open(nd.dentry, nd.mnt, flags);
628
629      return ERR_PTR(error);
630 }
```

这里的参数 flags 就是系统调用 open()传下来的，它遵循 open()界面上对 flags 的约定，但是这里调用的 open_namei()却对这些标志位有不同的约定（见 600 行至 613 行中的注释），所以要在调用 open_namei()前先加以变换，对于 i386 处理器，用于 open()界面上的标志位是在 include/asm-i386/fcntl.h 中定义的：

==================== include/asm-i386/fcntl.h 4 22 ====================

```
/* open/fcntl - O_SYNC is only implemented on blocks devices and on files
   located on an ext2 file system */
#define O_ACCMODE     0003
#define O_RDONLY       00
#define O_WRONLY       01
#define O_RDWR             02
#define O_CREAT          0100   /* not fcntl */
#define O_EXCL           0200   /* not fcntl */
#define O_NOCTTY      0400   /* not fcntl */
#define O_TRUNC         01000   /* not fcntl */
#define O_APPEND     02000
#define O_NONBLOCK   04000
#define O_NDELAY   O_NONBLOCK
#define O_SYNC         010000
#define FASYNC         020000   /* fcntl, for BSD compatibility */
#define O_DIRECT    040000   /* direct disk access hint - currently ignored */
#define O_LARGEFILE     0100000
#define O_DIRECTORY     0200000   /* must be a directory */
#define O_NOFOLLOW 0400000 /* don't follow links */
```

对于 flags 中最低两位所在的变换是由 620～621 行完成的，具体的变换如下：

00，表示无写要求，也就是“只读 ”　　变换成：01，表示要求读访问权。

01，表示“只写 ”　　变换成：10，表示要求写访问权。

10，表示“读和写 ”　　变换成：11，表示要求读和写访问权。

11，特殊，(O_RDWR | O_WRONLY)　　　　变换成：11，表示要求读和写访问权。

此外，如果 O_TRUNC 标志位为 1（表示要求截尾）则意味着要求写访问权。这些代码确实是很精练的。

下面就是调用 open_namei()，其代码在 fs/namei.c 中，我们分段来看。

==================== fs/namei.c 925 977 ====================

[sys_open()>filp_open()>open_namei()]

```
/*
 *   open_namei()
 *
 * namei for open - this is in fact almost the whole open-routine.
 *
 * Note that the low bits of "flag" aren't the same as in the open
 * system call - they are 00 - no permissions needed
 *               01 - read permission needed
 *               10 - write permission needed
 *               11 - read/write permissions needed
 * which is a lot more logical, and also allows the "no perm" needed
 * for symlinks (where the permissions are checked later).
 * SMP-safe
 */
int open_namei(const char * pathname, int flag, int mode, struct nameidata *nd)
{
     int acc_mode, error = 0;
     struct inode *inode;
     struct dentry *dentry;
     struct dentry *dir;
     int count = 0;

     acc_mode = ACC_MODE(flag);

     /*
      * The simplest case - just a plain lookup.
      */
     if (!(flag & O_CREAT)) {
          if (path_init(pathname, lookup_flags(flag), nd))
               error = path_walk(pathname, nd);
          if (error)
               return error;
          dentry = nd->dentry;
          goto ok;
     }

     /*
      * Create - we need to know the parent.
      */
     if (path_init(pathname, LOOKUP_PARENT, nd))
```

```
        error = path_walk(pathname, nd);
    if (error)
        return error;

    /*
     * We have the parent and last component. First of all, check
     * that we are not asked to creat(2) an obvious directory - that
     * will not do.
     */
    error = -EISDIR;
    if (nd->last_type != LAST_NORM || nd->last.name[nd->last.len])
        goto exit;

```

调用参数 flag 中的 O_CREAT 标志位表示如果要打开的文件不存在就创建这个文件。所以，如果这个标志位为 0，就仅仅是在文件系统中寻找目标节点，这就是通过 path_init()和 path_walk()根据目标节点的路径名找到该节点的 dentry 结构（以及 inode 结构）的过程，那已经在前面介绍过了。这里惟一值得一提的是在调用 path_init()时的参数 flag 还要通过 lookup_flags()进行一些处理，其代码也在 fs/namei.c 中：

==================== fs/namei.c 876 896 ====================

[sys_open()>flip_open()>open_namei()>lo0kup_flags()]

```
/*
 * Special case: O_CREAT|O_EXCL implies O_NOFOLLOW for security
 * reasons.
 *
 * O_DIRECTORY translates into forcing a directory lookup.
 */
static inline int lookup_flags(unsigned int f)
{
    unsigned long retval = LOOKUP_FOLLOW;

    if (f & O_NOFOLLOW)
        retval &= ~LOOKUP_FOLLOW;

    if ((f & (O_CREAT|O_EXCL)) == (O_CREAT|O_EXCL))
        retval &= ~LOOKUP_FOLLOW;

    if (f & O_DIRECTORY)
        retval |= LOOKUP_DIRECTORY;

    return retval;
}
```

可见，这里为上面 953 行的 path_init()设置了其参数 flags 中反映着搜索准则的标志位，但是最多只有 LOOKUP_FOLLOW 和 LOOKUP_DIRECTORY 两位有可能为 1。另一方面，在 open_namei()中对这些标志位的设置，即对 lookup_flags()的调用，是有条件的，仅在原先的参数 flag 中 O_CREAT 标志

位为 0 时才进行，所以我们知道这里 889 行中的条件一定是不成立的。如果 O_CREAT 标志位为 0，那么要是找不到目标节点就失败返回（而不创建这个节点）。

找到了目标节点的 dentry 结构以后，还要对其进行很多检验，等一下我们再回到这个话题。

如果 O_CREAT 标志位为 1，那就要复杂多了。首先也是通过 path_init()和 path_walk()沿着路径搜索，不过这一次寻找的不是目标节点的本身，而是其父节点，也就是目标文件所在的目录，所以在调用 path_init()时的标志位为 LOOKUP_PARENT。如果在搜索过程中出了错，例如某个中间节点不存在，或者不允许当前进程访问，那就出错返回了。否则，那就是找到了这个父节点。但是，找到了父节点并不表示整个路径就没有问题了。在正常的路径名中，路径的终点是一个文件名，此时 nameidata 结构中的 last_type 由 path_walk()设置成 LAST_NORM。但是，也有可能路径的终点为“ .”或“ ..”，也就是说路径的终点实际上是一个目录，此时 path_walk()将 last_type 设置成 LAST_DOT 或 LAST_DOTDOT（见“从路径名到目标节点”一节），那就应该视为出错而返回出错代码-EISDIR。这是为什么呢？因为 O_CREAT 标志位为 1 表示若目标节点不存在就创建该节点，可是 open()只能创建文件而不能创建目录，目录要由另一个系统调用 mkdir()来创建。同时，目标节点名必须是以“ \0”结尾的，那才是个正常的文件名。否则说明在目标节点名后面还有作为分隔符的“ /”字符，那么这还是个目录节点。注意虽然寻找的是目标节点的父节点，但是 path_walk()将 nameidata 结构中的 qstr 结构 last 设置成含有目标节点的节点名和字符串长度，只不过没有去寻找目标节点的 dentry 结构（以及 inode 结构）。读者不妨回过去重温一下 path_walk()的代码。

通过了这些检查，才说明真的找到了目标文件所在目录的 dentry 结构，可以往下执行了。继续往下看（namei.c）：

==================== fs/namei.c 978 1029 ====================

[sys_open()>filp_open()>open_namei()]

```
	dir = nd->dentry;
	down(&dir->d_inode->i_sem);
	dentry = lookup_hash(&nd->last, nd->dentry);

do_last:
	error = PTR_ERR(dentry);
	if (IS_ERR(dentry)) {
		up(&dir->d_inode->i_sem);
		goto exit;
	}

	/* Negative dentry, just create the file */
	if (!dentry->d_inode) {
		error = vfs_create(dir->d_inode, dentry, mode);
		up(&dir->d_inode->i_sem);
		dput(nd->dentry);
		nd->dentry = dentry;
		if (error)
			goto exit;
		/* Don't check for write permission, don't truncate */
		acc_mode = 0;
		flag &= ~O_TRUNC;
```

```
1000            goto ok;
1001        }
1002
1003        /*
1004         * It already exists.
1005         */
1006        up(&dir->d_inode->i_sem);
1007
1008        error = -EEXIST;
1009        if (flag & O_EXCL)
1010            goto exit_dput;
1011
1012        if (d_mountpoint(dentry)) {
1013            error = -ELOOP;
1014            if (flag & O_NOFOLLOW)
1015                goto exit_dput;
1016            do __follow_down(&nd->mnt,&dentry); while(d_mountpoint(dentry));
1017        }
1018        error = -ENOENT;
1019        if (!dentry->d_inode)
1020            goto exit_dput;
1021        if (dentry->d_inode->i_op && dentry->d_inode->i_op->follow_link)
1022            goto do_link;
1023
1024        dput(nd->dentry);
1025        nd->dentry = dentry;
1026        error = -EISDIR;
1027        if (dentry->d_inode && S_ISDIR(dentry->d_inode->i_mode))
1028            goto exit;
1029 ok:
```

我们已经找到了目标文件所在目录的 dentry 结构，并让指针 dir 指向这个数据结构，下一步就是通过 lookup_hash()寻找目标文件的 dentry 结构了（980 行）。函数 lookup_hash()的代码也在 fs/namei.c 中：

==================== fs/namei.c 704 748 ====================

[sys_open()>filp_open()>open_namei()>lookup_hash()]

```
/*
 * Restricted form of lookup. Doesn't follow links, single-component only,
 * needs parent already locked. Doesn't follow mounts.
 * SMP-safe.
 */
struct dentry * lookup_hash(struct qstr *name, struct dentry * base)
{
    struct dentry * dentry;
    struct inode *inode;
    int err;

    inode = base->d_inode;
    err = permission(inode, MAY_EXEC);
```

```
    dentry = ERR_PTR(err);
    if (err)
        goto out;

    /*
     * See if the low-level filesystem might want
     * to use its own hash..
     */
    if (base->d_op && base->d_op->d_hash) {
        err = base->d_op->d_hash(base, name);
        dentry = ERR_PTR(err);
        if (err < 0)
            goto out;
    }

    dentry = cached_lookup(base, name, 0);
    if (!dentry) {
        struct dentry *new = d_alloc(base, name);
        dentry = ERR_PTR(-ENOMEM);
        if (!new)
            goto out;
        lock_kernel();
        dentry = inode->i_op->lookup(inode, new);
        unlock_kernel();
        if (!dentry)
            dentry = new;
        else
            dput(new);
    }
out:
    return dentry;
}
```

这个函数先在 dentry 杂凑表队列中寻找，找不到就先创建一个新的 dentry 数据结构。再到目标文件所在的目录中寻找一次，如果找到了就把已经创建的 dentry 结构“归还”，找不到才采用它。为什么要采取这样的次序呢？这是因为在执行 d_alloc()的过程中有可能进入睡眠，这样就存在一种可能，就是当睡眠醒过来时情况已经变了。类似的情景读者在前几章也多次看到过。这样，从 lookup_hash()返回时，不管这目标文件存在与否，总是返回一个 dentry 数据结构指针（除非系统中的缓冲区已经用完）。如果目标文件的 dentry 结构原来就存在，那么结构中的 d_inode 指针指向该文件的 inode 结构；而如果这个 dentry 结构是新创建的，则其 d_inode 指针为 NULL，因为此时它还没有 inode 结构。

先看目标文件尚不存在的情况（见 990 行），那就要通过 vfs_create()创建。这个函数的代码比较长，却相对独立，并且受到多处调用，所以我们把它推迟到后面再来阅读，这里暂时只要知道这个函数的作用是创建文件就行了。

要是目标文件原来就存在呢？首先，在系统调用 open()的参数中标志位 O_CREAT 和 O_EXCL 同时为 1 表示目标文件在此之前必须不存在，所以如果已经存在就只好出错返回了，出错代码为-EEXIST。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

其次，这个目标文件有可能是个安装点，那样就要通过__follow_down()跑到所安装的文件系统中去，而且要通过一个 do-while 循环一直跑到头。此外，这个目标文件也有可能只是一个符号连接，而连接的对象有可能“悬空”，就是连接的对象在另一个设备上，但是那个设备却没有安装。所以代码中先检验目标节点是否符号连接，是的话就 goto 到 do_link 处，顺着连接前进到其目标节点，如果连接对象悬空则返回出错代码-ENOENT，否则便又 goto 回到 do_last 处对目标节点展开新一轮的检查。这一段代码在 open_namei()的最后，我们顺着程序的流程把它提到前面来阅读：

==================== fs/namei.c 1112 1152 ====================

[sys_open()>filp_open()>open_namei()]

```
do_link:
	error = -ELOOP;
	if (flag & O_NOFOLLOW)
		goto exit_dput;
	/*
	 * This is subtle. Instead of calling do_follow_link() we do the
	 * thing by hands. The reason is that this way we have zero link_count
	 * and path_walk() (called from ->follow_link) honoring LOOKUP_PARENT.
	 * After that we have the parent and last component, i.e.
	 * we are in the same situation as after the first path_walk().
	 * Well, almost - if the last component is normal we get its copy
	 * stored in nd->last.name and we will have to putname() it when we
	 * are done. Procfs-like symlinks just set LAST_BIND.
	 */
	UPDATE_ATIME(dentry->d_inode);
	error = dentry->d_inode->i_op->follow_link(dentry, nd);
	dput(dentry);
	if (error)
		return error;
	if (nd->last_type == LAST_BIND) {
		dentry = nd->dentry;
		goto ok;
	}
	error = -EISDIR;
	if (nd->last_type != LAST_NORM)
		goto exit;
	if (nd->last.name[nd->last.len]) {
		putname(nd->last.name);
		goto exit;
	}
	if (count++==32) {
		dentry = nd->dentry;
		putname(nd->last.name);
		goto ok;
	}
	dir = nd->dentry;
	down(&dir->d_inode->i_sem);
	dentry = lookup_hash(&nd->last, nd->dentry);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1150        putname(nd->last.name);
1151        goto do_last;
1152 }
```

读过 path_walk()的读者对这段代码不应该感到陌生。找到连接目标的操作主要是由具体文件系统在其 inode_operations 结构中的函数指针 follow_link 提供的。对于 Ext2 这个函数为 ext2_follow_link()，而最后这个函数又会通过 vfs_follow_link()调用 path_walk()，这读者已经看到过了。注意前面通过 path_init()设置在 nameidata 数据结构中的标志位并未改变，仍是 LOOKUP_PARENT。

对于目标节点是符号连接的情况，如果说在 follow_link 之前的“目标节点”是“视在目标节点”，那么 follow_link 以后的 nd->last 就是“真实目标节点”的节点名了，而 nd->dentry 则仍旧指向其父节点的 dentry 结构。

搜索到了真实目标节点的节点名后，同样还要检查这个节点是否只是个目录而不是文件（1136 行和 1138 行），如果是目录就要出错返回。至于搜索的结果为 LAST_BIND，则只发生于/proc 或类似的特殊文件系统中，我们在此并不关心。

最后，还要通过 lookup_hash()找到或创建真实目标节点的 dentry 结构，接着就转回到前面的 do_last 标号处，在那里又要重复前面对目标节点的一系列检验。检验的结果可能会发现我们所以为的“真实目标节点”实际上又是一个“视在目标节点”，那就又会转到这里的 do_link。为了防止陷入无穷循环，这里用了一个计数器 count 加以控制（见 1142 行），如果计数值达到了 32 就果断结束而转到标号 ok 处，在那里还要作进一步的检查，若发现仍是连接节点就会返回出错代码-ELOOP。读者以前在 path_walk()的代码中看到调用一个函数 do_follow_link()，在那里也有对连接链长度的控制（通过 task_struct 结构中的 link_count 字段），为什么这里还要另搞一套呢？这是因为 path_walk()对于旨在 LOOKUP_PARENT 的搜索只处理到目标节点的父节点为止，对目标节点本身是不作处理的。

只要连接链的长度不超出合理的范围，最终总会找到真正的目标节点。我们继续往下看（namei.c）：

==================== fs/namei.c 1029 1110 ====================

[sys_open()>filp_open()>open_namei()]

```
ok:
    error = -ENOENT;
    inode = dentry->d_inode;
    if (!inode)
        goto exit;

    error = -ELOOP;
    if (S_ISLNK(inode->i_mode))
        goto exit;

    error = -EISDIR;
    if (S_ISDIR(inode->i_mode) && (flag & FMODE_WRITE))
        goto exit;

    error = permission(inode,acc_mode);
    if (error)
        goto exit;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	/*
	 * FIFO's, sockets and device files are special: they don't
	 * actually live on the filesystem itself, and as such you
	 * can write to them even if the filesystem is read-only.
	 */
	if (S_ISFIFO(inode->i_mode) || S_ISSOCK(inode->i_mode)) {
	    flag &= ~O_TRUNC;
	} else if (S_ISBLK(inode->i_mode) || S_ISCHR(inode->i_mode)) {
	    error = -EACCES;
	    if (IS_NODEV(inode))
	        goto exit;

	    flag &= ~O_TRUNC;
	} else {
	    error = -EROFS;
	    if (IS_RDONLY(inode) && (flag & 2))
	        goto exit;
	}
	/*
	 * An append-only file must be opened in append mode for writing.
	 */
	error = -EPERM;
	if (IS_APPEND(inode)) {
	    if  ((flag & FMODE_WRITE) && !(flag & O_APPEND))
	        goto exit;
	    if (flag & O_TRUNC)
	        goto exit;
	}

	/*
	 * Ensure there are no outstanding leases on the file.
	 */
	error = get_lease(inode, flag);
	if (error)
	    goto exit;

	if (flag & O_TRUNC) {
	    error = get_write_access(inode);
	    if (error)
	        goto exit;

	    /*
	     * Refuse to truncate files with mandatory locks held on them.
	     */
	    error = locks_verify_locked(inode);
	    if (!error) {
	        DQUOT_INIT(inode);

	        error = do_truncate(dentry, 0);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1096          }
1097          put_write_access(inode);
1098          if (error)
1099              goto exit;
1100      } else
1101          if (flag & FMODE_WRITE)
1102              DQUOT_INIT(inode);
1103
1104      return 0;
1105
1106 exit_dput:
1107      dput(dentry);
1108 exit:
1109      path_release(nd);
1110      return error;
```

==================== fs/namei.c 1152 1152 ====================

```
1152 }
```

找到或者创建了目标文件以后，还要对代表着目标文件的数据结构进行一系列的检验，包括访问权限的检验以及一些处理（如截尾）。读者也许会对头三项检验感到奇怪，不是刚刚已经检查过了吗？回过去看一下就知道，那只是在特殊条件下进行的，而且并不完整。而现在，则是进行综合、完整的检验。

函数 permission()的代码读者已经在本章第三节中读过，这里的 acc_mode 是在一进入 open_namei()时就准备好了的（见 947 行），其中宏操作 ACC_MODE 及有关的定义分别在 fs/namei.c 和 include/asm-i386/fcntl.h 中给出：

==================== include/asm-i386/fcntl.h 6 6 ====================

```
6    #define O_ACCMODE     0003
```

==================== fs/namei.c 34 34 ====================

```
34   #define ACC_MODE(x) ("\000\004\002\006"[(x)&O_ACCMODE])
```

这里需要一些说明。“\000\004\002\006”是个字符串，其第一个字节为 8 进制的 0，第二个字节为 8 进制的 4，余类推。大家知道，在 C 语言中字符串与字符数组是等价的，所以这里把它作为字符数组来使用，而下标则为(flag & O_ACCMODE)。由于 O_ACCMODE 的值定义为 3，所以只有四种可能的下标值，即 0、1、2、3，具体取决于 flag。另一方面，前面讲过，在 filp_open()中调用 open_namei()之前已经对 flags 作了一些处理，将系统调用 open()界面上的标志位变换成了 open_namei()所要求的格式（见 filp_open()前面的注释）。但是，函数 permission()所要求的又有些不同，所以还要再变换一次。读者也许会说：“这不是存心让人看不懂吗？为什么要作这么多变换呢？”其实，这也是不得已的事，系统调用 open()界面上的标志位是从 Unix 的早期就定义好了的，要保持兼容就不能改变。所以，变换绝对是必须的事。可以斟酌的只是一次性变换还是分两次变换，读者仔细阅读 open_namei()的全部代码，就会体会到现在这样分两次变换还是合理的，尽管未必是惟一合理的方案。

下面还有一些对文件模式（由 inode 结构中的 i_mode 表示）与操作要求（由 flag 表示）之间的一致性的检验和处理。这些检验和处理大多数比较简单，例如对 FIFO 和插口文件就无所谓截尾，所以把 O_TRUNC 标志位清 0 等等。我们在这里并不关心磁盘空间配额（见 1093 行），所以集中看文件截尾。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

打开文件时，将 O_TRUNC 标志位设成 1 表示要将文件中原有的内容全部删除，称为“截尾”，这是由代码中的第 1083 行至 1100 行完成的。在此之前还调用了一个函数 get_lease()，这是与文件“租借”有关的操作，我们在此并不关心。

在文件的 inode 数据结构中有个计数器 i_writecount，用来对正在写访问该文件的进程计数。另一方面，有些文件可能已经通过 mmap()系统调用映射到某个进程的虚存空间，这个计数器就用于通过正常的文件操作和通过内存映射这两种写访问之间的互斥。当这个计数为负值时表示有进程可以通过虚存管理对文件进行写操作，为正值则表示某个或某些进程正在对文件进行写访问。内核中提供了 get_write_access()和 deny_write_access()两个函数来保证这种互斥访问，这两个函数的代码在 fs/namei.c 中：

==================== fs/namei.c 195 235 ====================

```
/*
 * get_write_access() gets write permission for a file.
 * put_write_access() releases this write permission.
 * This is used for regular files.
 * We cannot support write (and maybe mmap read-write shared) accesses and
 * MAP_DENYWRITE mmappings simultaneously. The i_writecount field of an inode
 * can have the following values:
 * 0: no writers, no VM_DENYWRITE mappings
 * < 0: (-i_writecount) vm_area_structs with VM_DENYWRITE set exist
 * > 0: (i_writecount) users are writing to the file.
 *
 * Normally we operate on that counter with atomic_{inc,dec} and it's safe
 * except for the cases where we don't hold i_writecount yet. Then we need to
 * use {get,deny}_write_access() - these functions check the sign and refuse
 * to do the change if sign is wrong. Exclusion between them is provided by
 * spinlock (arbitration_lock) and I'll rip the second arsehole to the first
 * who will try to move it in struct inode - just leave it here.
 */
static spinlock_t arbitration_lock = SPIN_LOCK_UNLOCKED;
int get_write_access(struct inode * inode)
{
     spin_lock(&arbitration_lock);
     if (atomic_read(&inode->i_writecount) < 0) {
          spin_unlock(&arbitration_lock);
          return -ETXTBSY;
     }
     atomic_inc(&inode->i_writecount);
     spin_unlock(&arbitration_lock);
     return 0;
}
int deny_write_access(struct file * file)
{
     spin_lock(&arbitration_lock);
     if (atomic_read(&file->f_dentry->d_inode->i_writecount) > 0) {
          spin_unlock(&arbitration_lock);
          return -ETXTBSY;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    }
    atomic_dec(&file->f_dentry->d_inode->i_writecount);
    spin_unlock(&arbitration_lock);
    return 0;
}
```

这里要注意，注释中所说的“ gets write permission”与通过 permission()检验当前进程对文件的写访问权限是两码事。前者是指对文件的常规写操作和对经过内存映射的文件内容的写操作之间的互斥，而后者则是出于安全性考虑的对访问权限的验证。

通过 get_write_access()得到了写操作的许可后，还要考虑目标文件是否已经被其它进程锁住了。从进程的角度考虑，对文件的每次读或写可以认为是“原子性”的，因为每次读或写都只要通过一次系统调用就可完成。但是，如果考虑连续几次的读、写操作，那么这些相继的读、写操作从总体上看就显然不是“原子性”的了。对于数据库一类的应用，这就成为一个问题。因此就发展起了对文件或文件中的某一部分内容“加锁”的技术。

“加锁”技术有两种。一种是由进程之间自己协调的，称为“协调锁”（advisory lock，或称 cooperative lock）。对于这一种锁，内核只提供加锁以及检测文件是否已经加锁的手段，即系统调用 flock()，但是内核并不参与锁的实施。也就是说，如果有进程不遵守“游戏规则”，不检查目标文件是否已经由别的进程加了锁就往里面写，内核是不加阻拦的，另一种则是由内核强制实施的，称为“强制锁”（mandatory lock），即使有进程不遵守游戏规则，不问三七二十一就要往加了锁的文件中写，内核也会加以阻拦。这种锁是对“协调锁”的改进与加强，具体通过 fcntl()系统调用实现。但是，在有些应用中并不适合使用强制锁，所以要给文件加上一个像开关一样的东西，这样才可以有选择地允许或不允许对一个文件使用强制锁。在 inode 结构的 i_flag 字段中定义的一个标志位 MS_MANDLOCK，就是起着开关的作用。这个标志位不仅用于 inode 结构，也用于 super_block 结构，对于整个文件子系统也可以在安装时将参数中的这个标志位设成 1 或 0，使整个设备上的文件全都允许或不允许使用强制锁。对于要求截尾的打开文件操作，内核应检查目标文件或者目标文件所在的设备是否允许使用强制锁，并且已经加了锁；如果已经加了强制锁便应该出错返回。这就是调用 inline 函数 locks_verify_locked()的原因，其代码在 include/linux/fs.h 中：

==================== include/linux/fs.h 892 904 ====================

[sys_open()>filp_open()>open_namei()>locks_verify_locked()]

```
/*
 * Candidates for mandatory locking have the setgid bit set
 * but no group execute bit -  an otherwise meaningless combination.
 */
#define MANDATORY_LOCK(inode) \
    (IS_MANDLOCK(inode) && ((inode)->i_mode & (S_ISGID | S_IXGRP)) == S_ISGID)

static inline int locks_verify_locked(struct inode *inode)
{
    if (MANDATORY_LOCK(inode))
        return locks_mandatory_locked(inode);
    return 0;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

有关的宏操作定义为：

==================== include/linux/fs.h 131 151 ====================

```
/*
 * Note that nosuid etc flags are inode-specific: setting some file-system
 * flags just means all the inodes inherit those flags by default. It might be
 * possible to override it selectively if you really wanted to with some
 * ioctl() that is not currently implemented.
 *
 * Exception: MS_RDONLY is always applied to the entire file system.
 *
 * Unfortunately, it is possible to change a filesystems flags with it mounted
 * with files in use.  This means that all of the inodes will not have their
 * i_flags updated.  Hence, i_flags no longer inherit the superblock mount
 * flags, so these have to be checked separately. -- rmk@arm.uk.linux.org
 */
#define __IS_FLG(inode,flg) ((inode)->i_sb->s_flags & (flg))

#define IS_RDONLY(inode) ((inode)->i_sb->s_flags & MS_RDONLY)
#define IS_NOSUID(inode)    __IS_FLG(inode, MS_NOSUID)
#define IS_NODEV(inode)     __IS_FLG(inode, MS_NODEV)
#define IS_NOEXEC(inode)    __IS_FLG(inode, MS_NOEXEC)
#define IS_SYNC(inode)      (__IS_FLG(inode, MS_SYNCHRONOUS) || ((inode)->i_flags & S_SYNC))
#define IS_MANDLOCK(inode)  __IS_FLG(inode, MS_MANDLOCK)
```

在老一些的版本中（从 Unix 系统 V 开始），曾经用 inode 结构的 mode 字段中 S_ISGID 和 S_IXGRP 两个标志位的结合来起 MS_MANDLOCK 标志位的作用。标志位 S_ISGID 与 S_ISUID 相似，表示在启动一个可执行文件时将进程的组号设置成该文件所属的组号。可是，如果这个文件对于所属的组根本就没有可执行属性（由 S_IXGRP 代表），那 S_ISGID 就失去了意义。所以，在正常情况下 S_ISGID 为 1 而 S_IXGRP 为 0 是自相矛盾的而不应该出现。既然如此，Unix 系统版本 V 就利用了这种组合来控制强制锁的使用。所以，只有在 inode 结构中或者 super_block 结构中的 MS_MANDLOCK 为 1，并且 inode 结构中的 S_ISGID 为 1 而 S_IXGRP 为 0 时才允许使用强制锁，这就是 901 行所表达的意思。

如果目标文件是允许使用强制锁的，那就要进一步检查是否已经加了锁。函数 locks_mandatory_locked()的代码在 fs/locks.c 中：

==================== fs/locks.c 675 692 ====================

[sys_open()>filp_open()>open_namei()>locks_verify_locked()>locks_mandatory_locked()]

```
int locks_mandatory_locked(struct inode *inode)
{
     fl_owner_t owner = current->files;
     struct file_lock *fl;

     /*
      * Search the lock list for this inode for any POSIX locks.
      */
     lock_kernel();
     for (fl = inode->i_flock; fl != NULL; fl = fl->fl_next) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (!(fl->fl_flags & FL_POSIX))
            continue;
        if (fl->fl_owner != owner)
            break;
    }
    unlock_kernel();
    return fl ? -EAGAIN : 0;
}
```

每个文件的 inode 结构中都有个 file_lock 数据结构队列 i_flock，每当一个进程对一个文件中的一个区间加锁时，就创建一个 file_lock 数据结构并将其挂入该文件的 inode 结构中。与 file_lock 结构有关的定义在 include/linux/fs.h 中：

==================== include/linux/fs.h 533 564 ====================

```
/*
 * The POSIX file lock owner is determined by
 * the "struct files_struct" in the thread group
 * (or NULL for no owner - BSD locks).
 *
 * Lockd stuffs a "host" pointer into this.
 */
typedef struct files_struct *fl_owner_t;

struct file_lock {
    struct file_lock *fl_next;  /* singly linked list for this inode  */
    struct list_head fl_link;   /* doubly linked list of all locks */
    struct list_head fl_block;  /* circular list of blocked processes */
    fl_owner_t fl_owner;
    unsigned int fl_pid;
    wait_queue_head_t fl_wait;
    struct file *fl_file;
    unsigned char fl_flags;
    unsigned char fl_type;
    loff_t fl_start;
    loff_t fl_end;

    void (*fl_notify)(struct file_lock *);    /* unblock callback */
    void (*fl_insert)(struct file_lock *);    /* lock insertion callback */
    void (*fl_remove)(struct file_lock *);    /* lock removal callback */

    struct fasync_struct * fl_fasync; /* for lease break notifications */

    union {
        struct nfs_lock_info    nfs_fl;
    } fl_u;
};
```

一个 file_lock 结构就是一把“锁”，结构中的指针 fl_file 指向目标文件的 file 结构，而 fl_start 和 fl_end 就确定了该文件中的一个区间，如果 fl_start 为 0 而 fl_end 为 OFFSET_MAX 就表示整个文件。此外，

fl_type 表示锁的性质，如读、写；fl_flags 中是一些标志位，这些标志位的定义也在 fs.h 中：

==================== include/linux/fs.h 526 531 ====================

```
#define FL_POSIX   1
#define FL_FLOCK   2
#define FL_BROKEN  4    /* broken flock() emulation */
#define FL_ACCESS  8    /* for processes suspended by mandatory locking */
#define FL_LOCKD   16   /* lock held by rpc.lockd */
#define FL_LEASE   32   /* lease held on this file */
```

标志位 FL_FLOCK 为 1 表示这个锁是通过传统的 flock()系统调用加上的，这种锁一定是协调锁，并且只能是对整个文件的。标志位 FL_POSIX 则表示通过 fcntl()系统调用加上的锁，它支持对文件中的区间加锁。由于在 POSIX 标准中规定了这种对文件中部分内容所加的锁，所以又称为 POSIX 锁。POSIX 锁可以是协调锁，也可以是强制锁，具体取决于前面所述的条件。这里，在 locks_mandatory_locked()的代码中检测的是强制锁，所以只关心 FL_POSIX 标志位为 1 的那些数据结构。

回到 open_namei()的代码中，如果目标文件并未加上强制锁，就可以通过 do_truncate()执行截尾了，其代码在 fs/open.c 中：

==================== fs/open.c 72 88 ====================

[sys_open()>filp_open()>open_namei()>do_truncate()]

```
int do_truncate(struct dentry *dentry, loff_t length)
{
	struct inode *inode = dentry->d_inode;
	int error;
	struct iattr newattrs;

	/* Not pretty: "inode->i_size" shouldn't really be signed. But it is. */
	if (length < 0)
		return -EINVAL;

	down(&inode->i_sem);
	newattrs.ia_size = length;
	newattrs.ia_valid = ATTR_SIZE | ATTR_CTIME;
	error = notify_change(dentry, &newattrs);
	up(&inode->i_sem);
	return error;
}
```

参数 length 为截尾后残留的长度，从 open_namei()中传下来的参数值为 0，表示全部切除。代码中先准备一个 iattr 结构，然后就通过 notify_change()来完成操作。函数 notify_change()的代码在 fs/attr.c 中：

==================== fs/attr.c 106 137 ====================

[sys_open()>filp_open()>open_namei()>do_truncate()>notify_change()]

```
int notify_change(struct dentry * dentry, struct iattr * attr)
{
	struct inode *inode = dentry->d_inode;
```

```
    int error;
    time_t now = CURRENT_TIME;
    unsigned int ia_valid = attr->ia_valid;

    if (!inode)
        BUG();

    attr->ia_ctime = now;
    if (!(ia_valid & ATTR_ATIME_SET))
        attr->ia_atime = now;
    if (!(ia_valid & ATTR_MTIME_SET))
        attr->ia_mtime = now;

    lock_kernel();
    if (inode->i_op && inode->i_op->setattr)
        error = inode->i_op->setattr(dentry, attr);
    else {
        error = inode_change_ok(inode, attr);
        if (!error)
            inode_setattr(inode, attr);
    }
    unlock_kernel();
    if (!error) {
        unsigned long dn_mask = setattr_mask(ia_valid);
        if (dn_mask)
            inode_dir_notify(dentry->d_parent->d_inode, dn_mask);
    }
    return error;
}
```

这里的目的是改变 inode 结构中的一些数据，以及执行伴随着这些改变的操作。这些操作常常是因文件系统而异的，所以具体的文件系统可以通过其 inode_operations 数据结构提供用于这个目的的函数（指针）。就 Ext2 文件系统而言，它并未提供这样的函数，所以通过 inode_change_ok()和 inode_setattr()两个函数来完成这个工作。两数 inode_change_ok()主要是对权限的检验，其代码在 fs/attr.c 中，我们把它留给读者自己阅读。

==================== fs/attr.c 17 58 ====================

[sys_open()>filp_open()>open_namei()>do_truncate()>notify_change()>inode_change_ok()]

```
/* POSIX UID/GID verification for setting inode attributes. */
int inode_change_ok(struct inode *inode, struct iattr *attr)
{
    int retval = -EPERM;
    unsigned int ia_valid = attr->ia_valid;

    /* If force is set do it anyway. */
    if (ia_valid & ATTR_FORCE)
        goto fine;

```

```
        /* Make sure a caller can chown. */
        if ((ia_valid & ATTR_UID) &&
            (current->fsuid != inode->i_uid ||
             attr->ia_uid != inode->i_uid) && !capable(CAP_CHOWN))
                goto error;

        /* Make sure caller can chgrp. */
        if ((ia_valid & ATTR_GID) &&
            (!in_group_p(attr->ia_gid) && attr->ia_gid != inode->i_gid) &&
            !capable(CAP_CHOWN))
                goto error;

        /* Make sure a caller can chmod. */
        if (ia_valid & ATTR_MODE) {
                if ((current->fsuid != inode->i_uid) && !capable(CAP_FOWNER))
                        goto error;
                /* Also check the setgid bit! */
                if (!in_group_p((ia_valid & ATTR_GID) ? attr->ia_gid :
                                inode->i_gid) && !capable(CAP_FSETID))
                        attr->ia_mode &= ~S_ISGID;
        }

        /* Check for setting the inode time. */
        if (ia_valid & (ATTR_MTIME_SET | ATTR_ATIME_SET)) {
                if (current->fsuid != inode->i_uid && !capable(CAP_FOWNER))
                        goto error;
        }
fine:
        retval = 0;
error:
        return retval;
}
```

函数 inode_setattr()的代码也在同一文件 attr.c 中：

==================== fs/attr.c 60 82 ====================

[sys_open()>filp_open()>open_namei()>do_truncate()>notify_change()>inode_setattr()]

```
void inode_setattr(struct inode * inode, struct iattr * attr)
{
        unsigned int ia_valid = attr->ia_valid;

        if (ia_valid & ATTR_UID)
                inode->i_uid = attr->ia_uid;
        if (ia_valid & ATTR_GID)
                inode->i_gid = attr->ia_gid;
        if (ia_valid & ATTR_SIZE)
                vmtruncate(inode, attr->ia_size);
        if (ia_valid & ATTR_ATIME)
                inode->i_atime = attr->ia_atime;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (ia_valid & ATTR_MTIME)
            inode->i_mtime = attr->ia_mtime;
        if (ia_valid & ATTR_CTIME)
            inode->i_ctime = attr->ia_ctime;
        if (ia_valid & ATTR_MODE) {
            inode->i_mode = attr->ia_mode;
            if (!in_group_p(inode->i_gid) && !capable(CAP_FSETID))
                inode->i_mode &= ~S_ISGID;
        }
        mark_inode_dirty(inode);
    }
```

我们在这里关心的只是文件大小的改变，这个改变及其伴随操作是通过 vmtruncate()完成的。函数 vmtruncate()所完成的操作以及它的代码都与文件的读写关系更为密切，所以我们把这个函数留到“文件的读写”一节中再来深入阅读。

最后，由于 inode 结构的内容改变了，所以要通过 mark_inode_dirty()把它挂到所属 super_block 结构的 s_dirty 队列中。

至此对目标文件所进行的操作都已完成，但是代表着当前进程与目标文件的连接的 file 结构却尚未建立。

返回到 filp_open()的代码中以后，下一个要调用的函数是 dentry_open()，它的任务就是建立起目标文件的一个“上下文”，即 file 数据结构，并让它与当前进程的 task_sturct 结构挂上钩，成为该文件驻在当前进程 task_struct 结构中的一个代表。函数 dentry_open()的代码在 fs/open.c 中：

==================== fs/open.c 632 679 ====================
[sys_open()>filp_open()>dentry_open()]

```
struct file *dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags)
{
    struct file * f;
    struct inode *inode;
    int error;

    error = -ENFILE;
    f = get_empty_filp();
    if (!f)
        goto cleanup_dentry;
    f->f_flags = flags;
    f->f_mode = (flags+1) & O_ACCMODE;
    inode = dentry->d_inode;
    if (f->f_mode & FMODE_WRITE) {
        error = get_write_access(inode);
        if (error)
            goto cleanup_file;
    }

    f->f_dentry = dentry;
    f->f_vfsmnt = mnt;
    f->f_pos = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	f->f_reada = 0;
	f->f_op = fops_get(inode->i_fop);
	if (inode->i_sb)
		file_move(f, &inode->i_sb->s_files);
	if (f->f_op && f->f_op->open) {
		error = f->f_op->open(inode,f);
		if (error)
			goto cleanup_all;
	}
	f->f_flags &= ~(O_CREAT | O_EXCL | O_NOCTTY | O_TRUNC);

	return f;

cleanup_all:
	fops_put(f->f_op);
	if (f->f_mode & FMODE_WRITE)
		put_write_access(inode);
	f->f_dentry = NULL;
	f->f_vfsmnt = NULL;
cleanup_file:
	put_filp(f);
cleanup_dentry:
	dput(dentry);
	mntput(mnt);
	return ERR_PTR(error);
}
```

顾名思义，get_empty_filp()的作用就是分配一个空闲的 file 数据结构。内核中有一个空闲 file 结构的队列 free_list，需要 file 结构时就从该队列中摘下一个，并将其暂时挂入一个中间队列 anon_list。在确认了对该文件可以进行写操作以后，就对这个空闲 file 结构进行初始化，然后通过 file_move()将其从中间队列脱链而挂入该文件所在设备的 super_block 结构中的 file 结构队列 s_files。

函数 get_write_access()一方面检查该文件是否因内存映射而不允许常规的写访问，另一方面如果允许常规的写访问就递增 inode 结构中的计数器 i_writecount，以此来保证常规的文件写访问与内存映射的文件写访问之间的互斥，其代码读者已在前面看到过。

读者已经熟知，file 结构中的指针 f_op 指向所属文件系统的 file_operations 数据结构。这个指针就是在这里设置的，而具体的 file_operations 结构指针则来自相应的 inode 结构。但是，如果相应文件系统是由可安装模块支持的就需要递增该模块的使用计数。这里的 fops_get()是个宏操作，定义于 include/linux/fs.h：

==================== include/linux/fs.h 859 863 ====================

```
/* Alas, no aliases. Too much hassle with bringing module.h everywhere */
#define fops_get(fops) \
	(((fops) && (fops)->owner)	\
		? ( try_inc_mod_count((fops)->owner) ? (fops) : NULL ) \
		: (fops))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

具体的文件系统还可能对打开文件有一些特殊的附加操作，如果有就通过其 file_operations 结构中的函数指针 open 提供。就 Ext2 文件系统而言，这个函数就是 ext2_open_file()，其代码在 fs/ext2/file.c 中：

==================== fs/ext2/file.c 82 94 ====================
[sys_open()>filp_open()>dentry_open()>ext2_open_file()]

```
/*
 * Called when an inode is about to be open.
 * We use this to disallow opening RW large files on 32bit systems if
 * the caller didn't specify O_LARGEFILE.  On 64bit systems we force
 * on this flag in sys_open.
 */
static int ext2_open_file (struct inode * inode, struct file * filp)
{
	if (!(filp->f_flags & O_LARGEFILE) &&
	    inode->i_size > 0x7FFFFFFFLL)
		return -EFBIG;
	return 0;
}
```

大家知道，inode 结构中有个 union，对于 Ext2 文件系统这个 union 被用作 ext2_inode_info 数据结构。在这个数据结构中有个字段 i_high_size，当文件的大小超过 32 位整数（2GB）时这个字段的值为文件大小的高 32 位。如果文件的大小真的超过 32 位，则在系统调用 open()的参数中要将一个标志位 O_LARGEFILE 置成 1，这是为 64 位系统结构考虑而设置的。在 inode 结构中另有一个表示文件大小的字段 i_size，其类型为 loff_t，实际上是 long long，即 64 位整数。如果 O_LARGEFILE 为 0，则文件大小必须小于 2GB。

最后，调用参数中的 O_CREAT，O_EXCL 等标志位已经不再需要了，因为它们只是对打开文件有作用，而现在文件已经打开，所以就把这些标志位清 0。

函数 dentry_open()返回指向新建立的 file 数据结构的指针。每进行一次成功的 open()系统调用就为目标文件建立起一个由 file 结构代表的上下文，而与该文件是否已经有其它的 file 结构无关。一个文件在内核中只能有一个 inode 结构，却可以有多个 file 结构。

这样，filp_open()的操作就完成了。回到 sys_open()的代码中，下面还有个 inline 函数。fd_install()的作用是将新建的 file 数据结构的指针“安装”到当前进程的 file_struct 结构中，确切地说是里面的已打开文件指针数组中，其位置即下标 fd，已经在前面分配好。该 inline 函数的代码在 include/linux/file.h 中：

==================== include/linux/file.h 74 96 ====================
[sys_open()>fd_install()]

```
/*
 * Install a file pointer in the fd array.
 *
 * The VFS is full of places where we drop the files lock between
 * setting the open_fds bitmap and installing the file in the file
 * array.  At any such point, we are vulnerable to a dup2() race
 * installing a file in the array before us.  We need to detect this and
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * fput() the struct file we are about to overwrite in this case.
 *
 * It should never happen - if we allow dup2() do it, _really_ bad things
 * will follow.
 */

static inline void fd_install(unsigned int fd, struct file * file)
{
	struct files_struct *files = current->files;

	write_lock(&files->file_lock);
	if (files->fd[fd])
		BUG();
	files->fd[fd] = file;
	write_unlock(&files->file_lock);
}
```

代码的作者加了注释，说明为什么不是简单地将指针 file 填入数组中指定的位置上，而要通过 xchg()把这个位置上原有的内容交换出来。如果交换出来的指针非 0 就说明已经有个 file 结构指针在这个位置上，所以要通过 fput()将其释放。从此，当前进程与目标文件之间就建立起了连接，可以通过这个特定的上下文访问该目标文件了。

看完了 sys_open()的主体，我们还要回过头去看一下 vfs_create()的代码，当系统调用 open()的参数中 O_CREAT 标志位为 1，而目标文件又不存在时，就要通过这个函数来创建，其代码在 fs/namei.c 中：

==================== fs/namei.c 898 923 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()]

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode)
{
	int error;

	mode &= S_IALLUGO & ~current->fs->umask;
	mode |= S_IFREG;

	down(&dir->i_zombie);
	error = may_create(dir, dentry);
	if (error)
		goto exit_lock;

	error = -EACCES;	/* shouldn't it be ENOSYS? */
	if (!dir->i_op || !dir->i_op->create)
		goto exit_lock;

	DQUOT_INIT(dir);
	lock_kernel();
	error = dir->i_op->create(dir, dentry, mode);
	unlock_kernel();
exit_lock:
	up(&dir->i_zombie);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
920      if (!error)
921           inode_dir_notify(dir, DN_CREATE);
922      return error;
923  }
```

参数 dir 指向所在目录的 inode 结构，而 dentry 则指向待创建文件的 dentry 结构。但是，此时待创建文件尚无 inode 结构，所以其 dentry 结构中的 d_inode 指针为 0，这样的 dentry 结构称为是“ negative ”的 dentry 结构。

每个进程都有个“文件访问权限屏蔽 ” umask，记录在其 fs_struct 结构中（task_struct 结构中的指针 fs 指向这个数据结构）。这是一些对于文件访问权限的屏蔽位，其格式与表示文件访问权限的 mode 相同。如果 umask 中的某一位为 1，则由此进程所创建的文件就把相应的访问权限“屏敞”掉。例如：如果一个进程的 umask 为 077，则由它所创建的文件只能由文件主使用，因为对同组人及其它用户的访问权限全给屏蔽掉了。进程的 umask 代代相传，但是可以通过系统调用 umask()加以改变。代码中第 902 行和第 903 行说明了怎样根据调用参数 mode 和进程的 umask 确定所创建文件的访问模式。这里的常数 S_IALLUGO 定义见文件 incluce/linux/stat.h：

===================== include/linux/stat.h 20 22 =====================

```
20   #define S_ISUID  0004000
21   #define S_ISGID  0002000
22   #define S_ISVTX  0001000
```

===================== include/linux/stat.h 50 51 =====================

```
50   #define S_IRWXUGO  (S_IRWXU|S_IRWXG|S_IRWXO)
51   #define S_IALLUGO  (S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO)
```

这些标志位的意义都已在“文件系统的访问权限与安全性”一节中介绍过。

不言而喻，创建文件时要改变其所在目录的内容。这个过程不容许受其它进程打扰，所以要放在临界区中完成。为此目的，在 inode 数据结构中提供了一个信号量 i_zombie。进入临界区后，先要检查当前进程的权限，看看是否允许在所在的目录中创建文件。代码中的 may_create()是个 inline 函数，其代码在 fs/namei.c 中：

===================== fs/namei.c 860 874 =====================

[sys_open()>filp_open()>open_namei()>vfs_create()>may_create()]

```
860  /*  Check whether we can create an object with dentry child in directory
861   *  dir.
862   *  1. We can't do it if child already exists (open has special treatment for
863   *     this case, but since we are inlined it's OK)
864   *  2. We can't do it if dir is read-only (done in permission())
865   *  3. We should have write and exec permissions on dir
866   *  4. We can't do it if dir is immutable (done in permission())
867   */
868  static inline int may_create(struct inode *dir, struct dentry *child) {
869      if (child->d_inode)
870           return -EEXIST;
871      if (IS_DEADDIR(dir))
872           return -ENOENT;
873      return permission(dir,MAY_WRITE | MAY_EXEC);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

这里 IS_DEADDIR()检查目标文件所在的目录是否实际上已被删除，其定义见文件 include/linux/fs.h：

==================== include/linux/fs.h 129 129 ====================

```
#define S_DEAD          32  /* removed, but still open directory */
```

==================== include/linux/fs.h 159 159 ====================

```
#define IS_DEADDIR(inode)   ((inode)->i_flags & S_DEAD)
```

删除文件时，要将所在目录 inode 结构中 i_flags 字段的 S_DEAD 标志位设成 1。但是，如果当时其 dentry 结构和 inode 结构的共享计数不能递减到 0 则不能将这两个数据结构释放。所以，存在着这样一种可能性，就是在当前进程通过 path_walk()找到了所在目录的 dentry 结构和 inode 结构之后，在试图进入由信号量 i_zombie 构成的临界区时进入了睡眠；而此时正在临界区中的另一个进程却删除了这个目录。不过，一旦当前进程进入了临界区以后，就再不会发生这种情况了。

通过了访问权限的检验以后，具体创建文件的操作则因文件系统而异，每种文件系统都通过其 inode_operations 结构提供用于创建文件的函数。就 Ext2 文件系统而言，这个函数为 ext2_create()，是在 fs/ext2/namei.c 中定义的：

==================== fs/ext2/namei.c 354 384 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()]

```
/*
 * By the time this is called, we already have created
 * the directory cache entry for the new file, but it
 * is so far negative - it has no inode.
 *
 * If the create succeeds, we fill in the inode information
 * with d_instantiate().
 */
static int ext2_create (struct inode * dir, struct dentry * dentry, int mode)
{
     struct inode * inode = ext2_new_inode (dir, mode);
     int err = PTR_ERR(inode);
     if (IS_ERR(inode))
          return err;

     inode->i_op = &ext2_file_inode_operations;
     inode->i_fop = &ext2_file_operations;
     inode->i_mapping->a_ops = &ext2_aops;
     inode->i_mode = mode;
     mark_inode_dirty(inode);
     err = ext2_add_entry (dir, dentry->d_name.name, dentry->d_name.len,
                   inode);
     if (err) {
          inode->i_nlink--;
          mark_inode_dirty(inode);
          iput (inode);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            return err;
        }
        d_instantiate(dentry, inode);
        return 0;
}
```

简言之，就是通过 ext2_new_inode()创建目标文件在存储设备上的索引节点和在内存中的 inode 结构，然后通过 ext2_add_entry()把目标文件的文件名与索引节点号写入其所在的目录（也是一个文件）中，最后由 d_instantiate()将目标文件的 dentry 结构和 inode 结构联系在一起。注意 inode 结构中的 i_op 和 i_fop 两个重要指针都是在这里设置的，还有其 a_ops 所指的 address_space 结构中的指针 a_ops 也是在这里设置的

先看 ext2_new_inode()，其代码在 fs/ext2/ialloc.c 中。因为比较长，我们分段来看。

==================== fs/ext2/ialloc.c 249 280 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()>ext2_new_inode()]

```
/*
 * There are two policies for allocating an inode.  If the new inode is
 * a directory, then a forward search is made for a block group with both
 * free space and a low directory-to-inode ratio; if that fails, then of
 * the groups with above-average free space, that group with the fewest
 * directories already is chosen.
 *
 * For other inodes, search forward from the parent directory\'s block
 * group to find a free inode.
 */
struct inode * ext2_new_inode (const struct inode * dir, int mode)
{
        struct super_block * sb;
        struct buffer_head * bh;
        struct buffer_head * bh2;
        int i, j, avefreei;
        struct inode * inode;
        int bitmap_nr;
        struct ext2_group_desc * gdp;
        struct ext2_group_desc * tmp;
        struct ext2_super_block * es;
        int err;

        /* Cannot create files in a deleted directory */
        if (!dir || !dir->i_nlink)
            return ERR_PTR(-EPERM);

        sb = dir->i_sb;
        inode = new_inode(sb);
        if (!inode)
            return ERR_PTR(-ENOMEM);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

参数 dir 指向所在目录的 inode 结构，这个结构中的 i_nlink 表示有几个目录项与这个 inode 结构相联系。

相对而言，在内存中分配一个 inode 结构是比较简单的，这里 J 的 new_inode()是个 inline 函数，定义于 include/linux/fs.h。

==================== include/linux/fs.h 1192 1200 ====================

```
static inline struct inode * new_inode(struct super_block *sb)
{
	struct inode *inode = get_empty_inode();
	if (inode) {
		inode->i_sb = sb;
		inode->i_dev = sb->s_dev;
	}
	return inode;
}
```

函数 get_empty_inode()将分配的空白 inode 结构挂入内核中的 inode_in_use 队列，其代码在 fs/inode.c 中，我们把它留给读者自己阅读。

下一步要做的是为目标文件在存储设备上分配一个索引节点。函数 ext2_new_inode 并不是单纯用来创建普通文件的，它也用来创建目录（当然，目录实际上也是文件，但是毕竟有些不同，目录是通过 mkdir()系统调用创建的）。实际上，调用这个函数的地方不光是 ext2_create()，还有 ext2_mknod()、ext2_mkdir()以及 ext2_symlink。代码的作者在函数的前面加了注释，说对于目录和普通文件采取了不同的分配策略。下面读者就会具体看到。

以前讲过，现代的块设备通常都是很大的。为了提高访问效率，就把存储介质划分成许多“块组”。一般来说，文件就应该与其所在目录存储在同一个块组中，这样才能提高效率。另一方面，文件的内容和文件的索引节点也应存储在同一块组中，所以在创建文件系统（格式化）时已经注意到了每个块组在索引节点和记录块数量之间的比例，这个比例是从统计信息得来的，取决于平均的文件大小。此外，根据统计，每一个块组中平均有多少个目录，也就是说每个目录中平均有多少文件，也大致上有个比例。所以，如果要创建的是文件，就应该首先考虑将它的索引节点分配在其所在目录所处的块组中。而如果要创建的是目录，则要考虑将来是否能将其属下的文件都容纳在同一块组中，所以应该找一个其空闲索引节点的数量超过整个设备上的平均值这么一个块组，而不惜离开其父节点所在的块组“另起炉灶”。了解了这些背景，读者应该可以读懂下面这段程序了。注意 282 行使指针 es 指向该文件系统超级块的缓冲区。沿着 ext2_new_inode()继续往下看（ialloc.c）：

==================== fs/ext2/ialloc.c 281 372 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()>ext2_new_inode()]

```
281		lock_super (sb);
282		es = sb->u.ext2_sb.s_es;
283	repeat:
284		gdp = NULL; i=0;
285	
286		if (S_ISDIR(mode)) {
287			avefreei = le32_to_cpu(es->s_free_inodes_count) /
288				sb->u.ext2_sb.s_groups_count;
```

```
/* I am not yet convinced that this next bit is necessary.
        i = dir->u.ext2_i.i_block_group;
        for (j = 0; j < sb->u.ext2_sb.s_groups_count; j++) {
            tmp = ext2_get_group_desc (sb, i, &bh2);
            if (tmp &&
                 (le16_to_cpu(tmp->bg_used_dirs_count) << 8) <
                  le16_to_cpu(tmp->bg_free_inodes_count)) {
                gdp = tmp;
                break;
            }
            else
            i = ++i % sb->u.ext2_sb.s_groups_count;
        }
*/
        if (!gdp) {
            for (j = 0; j < sb->u.ext2_sb.s_groups_count; j++) {
                tmp = ext2_get_group_desc (sb, j, &bh2);
                if (tmp &&
                   le16_to_cpu(tmp->bg_free_inodes_count) &&
                   le16_to_cpu(tmp->bg_free_inodes_count) >= avefreei) {
                    if (!gdp ||
                       (le16_to_cpu(tmp->bg_free_blocks_count) >
                        le16_to_cpu(gdp->bg_free_blocks_count))) {
                        i = j;
                        gdp = tmp;
                    }
                }
            }
        }
    }
    else
    {
        /*
         * Try to place the inode in its parent directory
         */
        i = dir->u.ext2_i.i_block_group;
        tmp = ext2_get_group_desc (sb, i, &bh2);
        if (tmp && le16_to_cpu(tmp->bg_free_inodes_count))
            gdp = tmp;
        else
        {
            /*
             * Use a quadratic hash to find a group with a
             * free inode
             */
            for (j = 1; j < sb->u.ext2_sb.s_groups_count; j <<= 1) {
                i += j;
                if (i >= sb->u.ext2_sb.s_groups_count)
                    i -= sb->u.ext2_sb.s_groups_count;
```

```
338                 tmp = ext2_get_group_desc (sb, i, &bh2);
339                 if (tmp &&
340                     le16_to_cpu(tmp->bg_free_inodes_count)) {
341                      gdp = tmp;
342                      break;
343                 }
344            }
345       }
346       if (!gdp) {
347            /*
348             * That failed: try linear search for a free inode
349             */
350            i = dir->u.ext2_i.i_block_group + 1;
351            for (j = 2; j < sb->u.ext2_sb.s_groups_count; j++) {
352                 if (++i >= sb->u.ext2_sb.s_groups_count)
353                      i = 0;
354                 tmp = ext2_get_group_desc (sb, i, &bh2);
355                 if (tmp &&
356                     le16_to_cpu(tmp->bg_free_inodes_count)) {
357                      gdp = tmp;
358                      break;
359                 }
360            }
361       }
362  }
363
364  err = -ENOSPC;
365  if (!gdp)
366       goto fail;
367
368  err = -EIO;
369  bitmap_nr = load_inode_bitmap (sb, i);
370  if (bitmap_nr < 0)
371       goto fail;
372
```

对于所创建目标为目录的情景，代码作者（不知是否原作者）把 290 行至 301 行暂时注销了，说不相信这是有必要的。不过，依我们看这倒是有好处的：如果一个块组里目录的数量与空闲索引节点的数量之比小于 1/256（见 293 行），则所创建目录（不包括其子目录）能够容纳在这个块组里的概率应该是很高的。这样做也有利于减少“另起炉灶”的次数，而让子目录尽量留在父目录所在的块组里。

确定了将索引节点分配在哪一个块组中以后，就要从该块组的索引节点位图中分配一个节点了（ialloc.c）：

==================== fs/ext2/ialloc.c 373 401 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()>ext2_new_inode()]

```
373  bh = sb->u.ext2_sb.s_inode_bitmap[bitmap_nr];
374  if ((j = ext2_find_first_zero_bit ((unsigned long *) bh->b_data,
375                        EXT2_INODES_PER_GROUP(sb))) <
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         EXT2_INODES_PER_GROUP(sb)) {
          if (ext2_set_bit (j, bh->b_data)) {
               ext2_error (sb, "ext2_new_inode",
                          "bit already set for inode %d", j);
               goto repeat;
          }
          mark_buffer_dirty(bh);
          if (sb->s_flags & MS_SYNCHRONOUS) {
               ll_rw_block (WRITE, 1, &bh);
               wait_on_buffer (bh);
          }
     } else {
          if (le16_to_cpu(gdp->bg_free_inodes_count) != 0) {
               ext2_error (sb, "ext2_new_inode",
                        "Free inodes count corrupted in group %d",
                        i);
               /* Is it really ENOSPC? */
               err = -ENOSPC;
               if (sb->s_flags & MS_RDONLY)
                    goto fail;

               gdp->bg_free_inodes_count = 0;
               mark_buffer_dirty(bh2);
          }
          goto repeat;
     }
```

在前一节中提到过，super_block 内的 ext2_sb_info 结构中有一个索引节点位图缓冲区的指针数组，用来缓冲存储若干个块组的位图。当需要使用某个块组的索引节点位图时，就先在这个数组中找，若找不到再从设备上把这个块组的位图读入缓冲区中，并让该数组中的某个指针指向这个缓冲区。这是由 load_inode_bitmap()完成的，其代码也在 fs/ext2/ialloc.c 中，我们把它留给读者自己阅读。

取得了目标块组的索引节点位图以后，就通过 ext2_find_first_zero_bit()从位图中找到一位仍然为 0 的位，也就是找到一个空闲的索引节点。一般情况下，这是不会失败的，因为该块组的描述结构已经告诉我们有空闲节点。

所谓从位图中分配一个索引节点，就是通过 ext2_set_bit()将其对应位设置成 1。这是一个宏操作，定义于 include/asm-i386/bitops.h

==================== include/asm-i386/bitops.h 248 248 ====================

```
#define ext2_set_bit                  __test_and_set_bit
```

也就是说，ext2_set_bit()一方面将位图中的某一位设成 1，另一方面还检查这一位原来是否为 1，如果是就说明有了冲突，因而要 goto 转回到标号 repeat 处另行寻找。否则，如果一切顺利，索引节点的分配就成功了，此时要立即把该索引节点所在记录块的缓冲区标志成“脏”。如果 super_block 结构的 s_flags 中的 MS_SYNCHRONOUS 标志位为 1，则立即要通过 ll_rw_block()把改变了的记录块写回磁盘并等待其完成。函数 ll_rw_block()的代码在 drivers/block/ll_rw_blk.c 中，这已经是属于设备驱动层的内容了，所以我们把它留给下册块设备驱动一章。

但是，尽管块组的描述结构告诉我们有空闲节点，ext2_find_first_zero_bit()还是有可能失败，因为块组的描述结构有可能已经损坏了，这往往发生在机器在运行时中途断电，或者用户不按规定程序关机的情况下。通常在这种情况发生后再次开机时系统会检测到文件系统“不干净”而强制进行一次文件系统检验（fsck）。并且，作为安全措施，在一个文件系统顺利安装了一定次数之后也要进行一次例行的检验。但尽管这样还是可能会有漏网之鱼，所以遇有块组描述结构中的信息与索引节点位图不一致时便说明块组已经损坏，该文件系统已经不一致了。如果发生了这样的情况（在位图中找不到空闲的索引节点），那就只好再找其它块组，所以（400 行）通过 goto 语句转回标号 repeat 处再来一次。

至此，变量 i 表示块组号，而 j 表示所分配的索引节点在本块组位图中的序号，根据这二者可以算出该节点在整个设备（文件子系统）中的索引节点号。我们继续往下看：

==================== fs/ext2/ialloc.c 402 466 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()>ext2_new_inode()]

```
402      j += i * EXT2_INODES_PER_GROUP(sb) + 1;
403      if (j < EXT2_FIRST_INO(sb) || j > le32_to_cpu(es->s_inodes_count)) {
404           ext2_error (sb, "ext2_new_inode",
405                   "reserved inode or inode > inodes count - "
406                   "block_group = %d,inode=%d", i, j);
407           err = -EIO;
408           goto fail;
409      }
410      gdp->bg_free_inodes_count =
411           cpu_to_le16(le16_to_cpu(gdp->bg_free_inodes_count) - 1);
412      if (S_ISDIR(mode))
413           gdp->bg_used_dirs_count =
414                cpu_to_le16(le16_to_cpu(gdp->bg_used_dirs_count) + 1);
415      mark_buffer_dirty(bh2);
416      es->s_free_inodes_count =
417           cpu_to_le32(le32_to_cpu(es->s_free_inodes_count) - 1);
418      mark_buffer_dirty(sb->u.ext2_sb.s_sbh);
419      sb->s_dirt = 1;
420      inode->i_mode = mode;
421      inode->i_uid = current->fsuid;
422      if (test_opt (sb, GRPID))
423           inode->i_gid = dir->i_gid;
424      else if (dir->i_mode & S_ISGID) {
425           inode->i_gid = dir->i_gid;
426           if (S_ISDIR(mode))
427                mode |= S_ISGID;
428      } else
429           inode->i_gid = current->fsgid;
430
431      inode->i_ino = j;
432      inode->i_blksize = PAGE_SIZE;    /* This is the optimal IO size (for stat), not the fs block size
*/
433      inode->i_blocks = 0;
434      inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
435      inode->u.ext2_i.i_new_inode = 1;
```

```
    inode->u.ext2_i.i_flags = dir->u.ext2_i.i_flags;
    if (S_ISLNK(mode))
        inode->u.ext2_i.i_flags &= ~(EXT2_IMMUTABLE_FL | EXT2_APPEND_FL);
    inode->u.ext2_i.i_faddr = 0;
    inode->u.ext2_i.i_frag_no = 0;
    inode->u.ext2_i.i_frag_size = 0;
    inode->u.ext2_i.i_file_acl = 0;
    inode->u.ext2_i.i_dir_acl = 0;
    inode->u.ext2_i.i_dtime = 0;
    inode->u.ext2_i.i_block_group = i;
    if (inode->u.ext2_i.i_flags & EXT2_SYNC_FL)
        inode->i_flags |= S_SYNC;
    insert_inode_hash(inode);
    inode->i_generation = event++;
    mark_inode_dirty(inode);

    unlock_super (sb);
    if(DQUOT_ALLOC_INODE(sb, inode)) {
        sb->dq_op->drop(inode);
        inode->i_nlink = 0;
        iput(inode);
        return ERR_PTR(-EDQUOT);
    }
    ext2_debug ("allocating inode %lu\n", inode->i_ino);
    return inode;

fail:
    unlock_super(sb);
    iput(inode);
    return ERR_PTR(err);
}
```

分配了空闲索引节点后，还要对其节点号作一次检查。Ext2 文件系统可能保留最初的若干索引节点不用，此外超级块中的 s_inodes_count 也可能与各块组中索引节点的总和不一致（通常发生在用户使用工具对超级块进行了某种修补以后）。

下面就是对块组描述结构和超级块中数据的调整，以及对新建立的 inode 结构的初始化了。读者应注意对新创建文件（或目录）的用户号 uid 和组号 gid 的设置。首先，新创文件的 uid 并不是当前进程的 uid，而是它的 fsuid。也就是说，如果当前进程是因为执行一个 suid 可执行程序而成为超级用户的，那么它所创建的文件属于超级用户（uid 为 0）。或者，如果当前进程通过设置进程的用户号转到了另一个用户的名下，那么它所创建的文件也就属于当前进程此时实际使用的用户号，即 fsuid。组号的情况也与此类似。但是安装文件系统时可以设置一个 GRPID 标志位，使得在该文件系统中新创文件时使用其所在目录的 gid，而不管当前进程的 fsgid 是什么。或者，如果虽然 GRPID 标志位为 0，但是，所在目录的模式中的 S_ISGID 标志为 1，也就继承其所在目录的 gid。

然后将新的 inode 结构链入到 inode_hashtable 中的某个杂凑队列里，insert_ionde_hash()的代码在 fs/inode.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/inode.c 796 812 ====================

[sys_open()>filp_open()>open_namei()>vfs_create()>ext2_create()>ext2_new_inode()>insert_inode_hash()]

```
/**
 *   insert_inode_hash - hash an inode
 *   @inode: unhashed inode
 *
 *   Add an inode to the inode hash for this superblock. If the inode
 *   has no superblock it is added to a separate anonymous chain.
 */

void insert_inode_hash(struct inode *inode)
{
     struct list_head *head = &anon_hash_chain;
     if (inode->i_sb)
          head = inode_hashtable + hash(inode->i_sb, inode->i_ino);
     spin_lock(&inode_lock);
     list_add(&inode->i_hash, head);
     spin_unlock(&inode_lock);
}
```

索引节点号只在同一设备上保持惟一性，所以在杂凑计算时将所在设备的 super_block 结构的地址也一起计算进去，以保证其全系统范围的惟一性。

由于我们并不关心设备上存储空间的配额问题，ext2_new_inode()的操作就完成了。回到 ext2_create()的代码中，接着是设置新创 inode 结构中的 inode_operations 结构指针和 file 结构指针，还有用于文件映射（至虚存空间中）的 address_space_operations 结构指针，使它们一一指向由 Ext2 文件系统提供的相应数据结构。这样，对这个新建文件，VFS 层与 Ext2 层之间的界面就设置好了。这些指针决定了对该文件所作的一些文件操作要通过由 Ext2 文件系统所提供的函数来完成。

至此，新文件的索引节点已经分配，内核中的 inode 数据结构也已经建立并设置好。由于新的 inode 已经通过 mark_inode_dirty()设置成“脏”，并从杂凑队列转移到了 super_block 结构中的 s_dirty 队列里，这样内核就会（在适当的时机）把这个 inode 结构的内容写回设备上的索引节点，因此可以认为文件本身的创建已经完成了。但是，尽管如此，这个文件还只是一个“孤岛”，通向这个文件的路径还不存在。所以，回到 ext2_create()中，下一步是要在该文件所在的目录中增加一个目录项，使新文件的文件名与其索引节点号挂上钩并出现在目录中，从而建立起通向这个文件的路径，这是由 ext2_add_entry()完成的，其代码在 fs/ext2/namei.c 中。如前所述，目录实际上也是文件，所以在目录中增加一个目录项的操作就与普通文件的读 / 写很相似，我们建议读者在学习了下一节“文件的读与写”以后回过头来自己读一下这段代码。

最后，还要让新建文件的 dentry 结构（在 open_namei()中由 lookup_hash()创建）与 inode 结构之间也挂上钩，这是由 d_instantiate()完成的，其代码已在前面读过了。

函数 ext2_create()执行完毕以后，vfs_create()的任务也就完成了。

看完了文件的打开，再来看看文件的关闭。系统调用 close()是由内核中的 sys_close()实现的，有关的代码基本上都在 fs/open.c 中：

==================== fs/open.c 810 835 ====================

```
/*
 * Careful here! We test whether the file pointer is NULL before
 * releasing the fd. This ensures that one clone task can't release
 * an fd while another clone is opening it.
 */
asmlinkage long sys_close(unsigned int fd)
{
	struct file * filp;
	struct files_struct *files = current->files;

	write_lock(&files->file_lock);
	if (fd >= files->max_fds)
		goto out_unlock;
	filp = files->fd[fd];
	if (!filp)
		goto out_unlock;
	files->fd[fd] = NULL;
	FD_CLR(fd, files->close_on_exec);
	__put_unused_fd(files, fd);
	write_unlock(&files->file_lock);
	return filp_close(filp, files);

out_unlock:
	write_unlock(&files->file_lock);
	return -EBADF;
}
```

代码中 FD_CLR 以及有关的宏操作定义如下，分别在 time.h 和 include/asm-i386/posix_types.h 中：

==================== include/linux/time.h 109 109 ====================

```
#define FD_CLR(fd,fdsetp)    __FD_CLR(fd,fdsetp)
```

==================== include/asm-i386/posix_types.h 55 57 ====================

```
#define __FD_CLR(fd,fdsetp) \
		__asm__ __volatile__("btrl %1,%0": \
			"=m" (*(__kernel_fd_set *) (fdsetp)):"r" ((int) (fd)))
```

它将位图 files->close_on_exec 中序号为 fd 的那一位清成 0。

函数__put_unused_fd()的代码在 fs/open.c 中：

==================== include/linux/file.h 58 63 ====================

[sys_close()>__put_unused_fd()]

```
static inline void __put_unused_fd(struct files_struct *files, unsigned int fd)
{
	FD_CLR(fd, files->open_fds);
	if (fd < files->next_fd)
		files->next_fd = fd;
}
```

代码的作者在 sys_close()的注释中讲述了在释放打开文件号之前先检查与其对应的 file 结构指针

PDF-XCHANGE Click to buy NOW! www.docu-track.com

filp 是否为 0 的重要性。一方面这是因为在打开文件时分配打开文件号在前，而“安装 ” file 结构指针在最后（见 sys_open()的代码）。另一方面，一个进程在 fork 子进程时可以选择让子进程共享而不是“继承”它的资源，包括其 files_srtuct 结构。这样，如果两个进程共享同一个 file_struct 结构，其中一个进程正在打开文件，已经分配了打开文件号，但是尚未安装 file 结构指针，而另一个进程却在中途挤进来关闭这个“已打开文件”而释放了这个打开文件号，那当然会造成问题。

然后，就像 sys_open()的主体是 filp_open()一样，sys_close()的主体也是 filp_close()，其代码也在 fs/open.c 中：

==================== fs/open.c 786 808 ====================

[sys_close()>filp_close()]

```
/*
 * "id" is the POSIX thread ID. We use the
 * files pointer for this..
 */
int filp_close(struct file *filp, fl_owner_t id)
{
     int retval;

     if (!file_count(filp)) {
          printk("VFS: Close: file count is 0\n");
          return 0;
     }
     retval = 0;
     if (filp->f_op && filp->f_op->flush) {
          lock_kernel();
          retval = filp->f_op->flush(filp);
          unlock_kernel();
     }
     fcntl_dirnotify(0, filp, 0);
     locks_remove_posix(filp, id);
     fput(filp);
     return retval;
}
```

有些文件系统安排在关闭文件时“冲刷”文件的内容，即把文件中已经改变过的内容写回设备上，并因而在其 file_operations 数据结构中提供相应的函数指针 flush。不过，Ext2 并不作这样的安排，其函数指针 flush 为空指针，这一来关闭文件的操作就变得简单了。此外，当前进程可能对欲关闭的文件加了 POSIX 锁，但是忘了在关闭前把锁解除，所以调用 locks_remove_posix()试一下，以防万一。

最后，就是 fput()了。它递减 file 结构中的共享计数，如果递减后达到了 0 就释放该 file 结构，有关的代码在 include/linux/fs.h 和 fs/file_table.c 中。读者应注意从 sys_close()开始我们并未见到与 fput()配对的 fget()。其实，这个计数是当初在打开文件时在 get_empty_filp()中设置成 1 的，所以这里的递减与此遥相呼应。至于这一次 fput()是否能使该计数达到 0，则取决于此时是否还有别的活动或进程在共享这个数据结构。例如，要是当初打开这个文件的进程，clone()了一个线程，那就会在 clone()的时候递增这个计数，如果所创建的线程尚未关闭这个文件，则因共享计数大于 1 而不会递减至 0。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/file_table.c 99 123 ====================

[sys_close()>filp_close()>fput()]

```
void fput(struct file * file)
{
	struct dentry * dentry = file->f_dentry;
	struct vfsmount * mnt = file->f_vfsmnt;
	struct inode * inode = dentry->d_inode;

	if (atomic_dec_and_test(&file->f_count)) {
		locks_remove_flock(file);
		if (file->f_op && file->f_op->release)
			file->f_op->release(inode, file);
		fops_put(file->f_op);
		file->f_dentry = NULL;
		file->f_vfsmnt = NULL;
		if (file->f_mode & FMODE_WRITE)
			put_write_access(inode);
		dput(dentry);
		if (mnt)
			mntput(mnt);
		file_list_lock();
		list_del(&file->f_list);
		list_add(&file->f_list, &free_list);
		files_stat.nr_free_files++;
		file_list_unlock();
	}
}
```

在 fput()中又来处理当前进程可能已经对目标文件加上而未及解除的锁，但是这一次关心的是 FL_FLOCK 锁。如前所述，这种锁一定是“协调锁”；而刚才处理的是 POSIX 锁，它可以是协调锁也可以是强制锁。

代码中的 fops_put()是个宏操作：

==================== include/linux/fs.h 865 869 ====================

```
#define fops_put(fops) \
do { \
	if ((fops) && (fops)->owner) \
		__MOD_DEC_USE_COUNT((fops)->owner);	\
} while(0)
```

显然，这里关心的是动态安装模块的使用计数。

此外，每种文件系统可以对 file 结构的释放规定一些附加操作，通过其 file_operations 结构中的函数指针 release 提供相应的操作，如果这个指针非 0 就表示需要调用这个函数。就 Ext2 文件系统而言，这个函数是 ext2_release_file()，其代码在 fs/ext2/file.c 中：

==================== fs/ext2/file.c 70 80 ====================

[sys_close()>filp_close()>fput()>ext2_release_file()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * Called when an inode is released. Note that this is different
 * from ext2_file_open: open gets called at every open, but release
 * gets called only when /all/ the files are closed.
 */
static int ext2_release_file (struct inode * inode, struct file * filp)
{
	if (filp->f_mode & FMODE_WRITE)
		ext2_discard_prealloc (inode);
	return 0;
}
```

操作很简单，只是把预分配的数据块（见下一节）释放掉而已。

把 file 结构释放以后，目标文件的 dentry 结构以及所在设备的 vfsmount 结构就少了一个用户，所以还要调用 dput()和 mntput()递减它们的共享计数。同样，如果递减后达到了 0 就要将数据结构释放。还有，如果当初打开这个文件时的模式为写访问，则还要通过 put_write_access()递减其 inode 结构中的 i_writecount 计数。如前所述，这个计数用于按普通的文件操作与按内存映射访问文件这两种途径间的互斥。

最后，所谓“释放 ” file 结构，就是把它从 inode_hashtabte 中的杂凑队列里脱链，退还到 free_list 中。

5.6　文件的写与读

只有在“打开”了文件以后，或者说建立起进程与文件之间的“连接”之后，才能对文件进行读／写。文件的读／写主要是通过系统调用 read()和 write()完成的，对于读／写文件的进程，目标文件由一个“打开文件号”代表。

为了提高效率，稍为复杂一些的操作系统对文件的读／写都是带缓冲的，Linux 当然也不例外。像 VFS 一样，Linux 文件系统的缓冲机制也是它的一大特色。所谓缓冲，是指系统为最近刚读／写过的文件内容在内核中保留一份副本，以便当再次需要已经缓冲存储在副本中的内容时就不必再临时从设备上读入，而需要写的时候则可以先写到副本中，待系统较为空闲时再从副本写入设备。在多进程的系统中，由于同一文件可能为多个进程所共享，缓冲的作用就更显著了。

然而，怎样实现缓冲，在哪一个层次上实现缓冲，却是一个值得仔细加以考虑的问题。回顾一下本章开头处的文件系统层次图（图 5.3 和图 5.1），在系统中处于最高层的是进程，这一层可以称为“应用层”，是在用户空间运行的，在这里代表着目标文件的是“打开文件号”。在这一层中提供缓冲似乎最贴近文件内容的使用者，但是那样就需要用户进程的介入，从而不能做到对使用者“透明”，并且缓冲的内容不能为其它进程所共享，所以显然是不妥当的。在应用层以下是“文件层”，又可细分为 VFS 层和具体的文件系统层，再下面就是“设备层”了。这些层次都在内核中，所以在这些层次上实现缓冲都可以达到对用户透明的目标。设备层是最贴近设备，即文件内容的“源头”的地方，在这里实现缓冲显然是可行的。事实上，早期 Unix 内核中的文件缓冲就是以数据块缓冲的形式在这一层上实现的。但是，设备层上的缓冲离使用者的距离太远了一点，特别是当文件层又分为 VFS 和具体文件系统两个

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

子层时，每次读 / 写都要穿越这么多界面深入到设备层就难免使人有一种“长途跋涉”之感。很自然地，设计人员把眼光投向了文件层。

在文件层中有三种主要的数据结构，就是 file 结构、dentry 结构以及 inode 结构。

先看 file 结构。前面讲过，一个 file 结构代表着目标文件的一个上下文，不但不同的进程可以在同一个文件上建立不同的上下文，就是同一个进程也可以通过打开同一个文件多次而建立起多个上下文。如果在 file 结构中设置一个缓冲区队列，那么缓冲区中的内容虽然贴近这个特定上下文的使用者，却不便于为多个进程共享，甚至不便于同一个进程打开的不同上下文“共享”。这显然是不合适的，需要把这些缓冲区像数学上的“提取公因子”那样放到一个公共的地方。

那么 dentry 结构怎么样？这个数据结构并不属于某一个上下文，也不属于某一个进程，可以为所有的进程和上下文共享。可是，dentry 结构与目标文件并不是一对一的关系，通过文件连接，我们可以为已经存在的文件建立“别名”。一个 dentry 结构只是惟一地代表着文件系统中的一个节点，也就是一个路径名，但是多个节点可以同时代表着同一个文件，所以，还应该再来一次“提取公因子”。

显然，在 inode 数据结构中设置一个缓冲区队列是最合适不过的了，首先，inode 结构与文件是一对一的关系，即使一个文件有多个路径名，最后也归结到同一个 inode 结构上。再说，一个文件中的内容是不能由其它文件共享的，在同一时间里，设备上的每一个记录块都只能属于至多一个文件（或者就是空闲），将载有同一个文件内容的缓冲区都放在其所属文件的 inode 结构中是很自然的事。因此，在 inode 数据结构中设置了一个指针 i_mapping，它指向一个 address_space 数据结构（通常这个数据结构就是 inode 结构中的 i_data），缓冲区队列就在这个数据结构中。

不过，挂在缓冲区队列中的并不是记录块而是内存页面。也就是说，文件的内容并不是以记录块为单位，而是以页面为单位进行缓冲的。如果记录块的大小为 1K 字节，那么一个页面就相当于 4 个记录块。为什么要这样做呢？这是为了将文件内容的缓冲与文件的内存映射结合在一起。我们在第 2 章中提到过，一个进程可以通过系统调用 mmap()将一个文件映射到它的用户空间。建立了这样的映射以后，就可以像访问内存一样地访问这个文件。如果将文件的内容以页面为单位缓冲，放在附属于该文件的 inode 结构的缓冲队列中，那么只要相应地设置进程的内存映射表，就可以很自然地将这些缓冲页面映射到进程的用户空间中。这样，在按常规的文件操作访问一个文件时，可以通过 read()和 write()系统调用目标文件的 inode 结构访问这些缓冲页面；而通过内存映射机制访问这个文件时，就可以经由页面映射表直接读写这些缓冲着的页面。当目标页面不在内存中时，常规的文件操作通过系统调用 read()、write()的底层将其从设备上读入，而通过内存映射机制访问这个文件时则由“缺页异常”的服务程序将目标页面从设备上读入。也就是说，同一个缓冲页面可以满足两方面的要求，文件系统的缓冲机制和文件的内存映射机制巧妙地结合在一起了。明白了这个背景，对于上述的指针为什么叫 i_mapping，它所指向的数据结构为什么叫 address_space，就不会感到奇怪了。

可是，尽管以页面为单位的缓冲对于文件层确实是很好的选择，对于设备层则不那么合适了。对设备层而言，最自然的当然还是以记录块为单位的缓冲，因为设备的读 / 写都是以记录块为单位的。不过，从磁盘上读 / 写时主要的时间都花在准备工作（如磁头组的定位）上，一旦准备好了以后读一个记录块与接连读几个记录块相差不大，而且每次只读写一个记录块倒反而是不经济的。所以每次读写若干连续的记录块、以页面为单位来缓冲也并不成为问题。另一方面，如果以页面为单位缓冲，而一个页面相当于若干个记录块，那么无论是对于缓冲页面还是对于记录块缓冲区，其控制和附加信息（如链接指针等）显然应该游离于该页面之外，这些信息不应该映射到进程的用户空间。这个问题也

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

不难解决。读者不妨回顾一下，第 2 章中讲过的 page 数据结构就是这样。在 page 数据结构中有个指针 virtual 指向其所代表的页面，但是 page 结构本身则不在这个页面中。同样地，在“缓冲区头部”即 buffer_head 数据结构中有一个指针 b_data 指向缓冲区，而 buffer_head 结构本身则不在缓冲区中。所以，在设备层中只要保持一些 buffer_head 结构，让它们的 b_data 指针分别指向缓冲页面中的相应位置上就可以了。以一个缓冲页面为例，在文件层它通过一个 page 数据结构挂入所属 inode 结构的缓冲页面队列，并且同时又可以通过各个进程的页面映射表映射到这些进程的内存空间；而在设备层则又通过若干（通常是四个，因为页面的大小为 4KB，而缓冲区的大小为 1KB）buffer_head 结构挂入其所在设备的缓冲区队列。这样，以页面为单位为文件内容建立缓冲真是“一箭三雕”。下页的示意图（图 5.6）也许有助于读者对缓冲机制的理解。

在这样一个结构框架中，一旦所欲访问的内容已经在缓冲页面队列中，读文件的效率就很高了，只要找到文件的 inode 结构（file 结构中有指针指向 dentry 结构，而 dentry 结构中有指针指向 inode 结构）就找到了缓冲页面队列，从队列中找到相应的页面就可以读出了。缓冲页面的 page 结构除链入附属于 inode 结构的缓冲页面队列外，同时也链入到一个杂凑表 page_hash_table 中的杂凑队列中（图中没有画出），所以寻找目标页面的操作也是效率很高的，并不需要在整个缓冲页面队列中线性搜索。

那么，写操作又如何呢？如前所述，一旦目标记录块已经存在于缓冲页面中，写操作只是把内容写到该缓冲页面中，所以从发动写操作的进程的角度来看速度也是很快的。至于改变了内容的缓冲页面，则由系统负责在 CPU 较为空闲时写入设备。为了这个目的，内核中设置了一个内核线程 kflushd。平时这个线程总是在睡眠，有需要时（例如写操作以后）就将其唤醒，然后当 CPU 较为空闲时就会调度其运行，将已经改变了内容的缓冲页面写回设备上。这样，启动写操作的进程和 kflushd 就好像是一条流水作业线上的上下两个工位上的操作工，而改变缓冲页面的内容（写操作）与将改变了内容的缓冲页面写回设备上（称为“同步”）则好像是上下两道工序。除这样的“分工合作”，以外，每个打开了某个文件的进程还可以直接通过系统调用 sync()强行将缓冲页面写回设备上。此外，缓冲页面的 page 结构还链入到一个 LRU 队列中，要是一个页面很久没有受到访问，内存空间又比较短缺，就可以把它释放而另作他用。

除通过缓冲来提高文件读 / 写的效率外，还有个措施是“预读”。就是说，如果一个进程发动了对某一个缓冲页面的读（或写）操作，并且该页面尚不在内存中而需要从设备上读入，那么就可以预测，通常情况下它接下去可能会继续往下读写，因此不妨预先将后面几个页面也一起读进来。如前所述，对于磁盘一类的“块设备”，读操作中最费时间的是磁头组定位，一旦到了位，从设备多读几个记录块并不相差多少时间。一般而言，对文件的访问有两种形式。一种是“随机访问”，其访问的位置并无规律；另一种是“顺序访问”。预读之所以可能提高效率就是因为大量的文件操作都是顺序访问。其实，以页面（而不是记录块）为单位的缓冲本身就隐含着预读，因为通常一个页面包含着 4 个记录块，只要访问的位置不在其最后一个记录块中，就多少要预读几个记录块，只不过预读的量很小而已。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

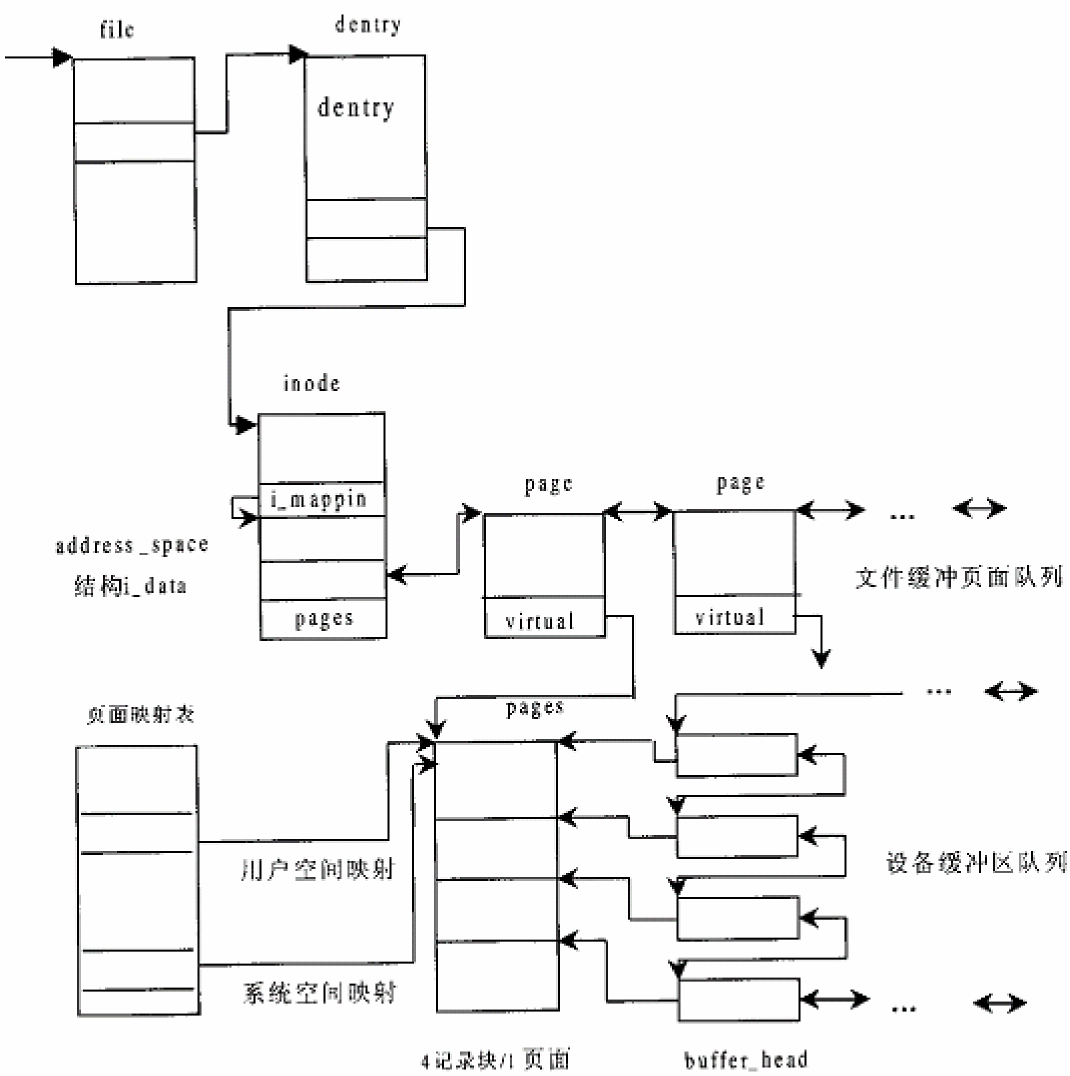


图 5.6　文件页面缓冲队列与设备缓冲区队列的联系图

在早期的 Unix 系统中，由于当时的磁盘容量小，速度慢，内存也小，一般只预读一个记录块。而现在的预读，则动辍就是几十 K 字节，甚至上百 K 字节。当然，那也要视具体情况而定，所以在 include/linux/blkdev.h 中定义了一个常数 MAX_READAHEAD，其定义为：

===================== include/linux/blkdev.h 184 186 ====================

```
/* read-ahead in pages.. */
#define MAX_READAHEAD   31
#define MIN_READAHEAD   3
```

这里的数值 31 表示 31 个页面，即 124K 字节。从这里也可以看出，许多比较小的文件其实都是一次就全部预读入内存的。当然，这里说的是最大预读量，实际运行时还要看其它条件，未必真的预读

PDF-XCHANGE Click to buy NOW! www.docu-track.com

那么多。

由于预读的提前量已经不再限于一个记录块，现在 file 结构中实际上要维持两个上下文了。一个就是由“当前位置”f_pos 代表的真正的读 / 写上下文，而另一个则是预读的上下文。为此目的，在 file 结构中增设了 f_reada、f_ramax、f_raend、f_rawin 等几个字段。这几个字段的名称反映了它们的用途（ra 表示“ read ahead”），具体的含义在下面的代码中就可看到。

另一方面，预读虽然并不花费很多时间，但毕竟还是需要一点时间。当一个进程启动一次对文件内容的访问，而访问的目标又恰好不在内存中因而需要从设备上读入时，该进程只好暂时交出运行权，进入睡眠中等待，称之为“受阻”（blocked）。可是等待多久呢？一旦本次访问的目标页面进入了内存，等待中的进程就可以而且应该恢复运行了，而没有理由等待到所有预读的页面也全部进入内存。从设备上读 / 写一般都是通过 DMA 进行的，它固然需要一定的时间，但是并不需要 CPU 太多的干预，CPU 完全可以忙自己的事。所以，从设备上读入的操作可以分成两部分。第一部分是必须要等待的，在此期间启动本次操作的进程只好暂时停下来，这一部分操作是“同步”的。第二部分则无须等待，在此期间启动本次操作的进程可以继续运行，所以这一部分操作是异步的。至于写操作，则如前所述在大多数情况下是留给内核线程 kflushd 完成的，那当然是异步的。

读完了上面这一大段的概述，现在可以开始读代码了。先看 sys_write()，这是系统调用 write()在内核中的实现，其代码在 fs/read_write.c 中：

==================== fs/read_write.c 144 169 ====================

```
asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
{
	ssize_t ret;
	struct file * file;

	ret = -EBADF;
	file = fget(fd);
	if (file) {
		if (file->f_mode & FMODE_WRITE) {
			struct inode *inode = file->f_dentry->d_inode;
			ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file,
				file->f_pos, count);
			if (!ret) {
				ssize_t (*write)(struct file *, const char *, size_t, loff_t *);
				ret = -EINVAL;
				if (file->f_op && (write = file->f_op->write) != NULL)
					ret = write(file, buf, count, &file->f_pos);
			}
		}
		if (ret > 0)
			inode_dir_notify(file->f_dentry->d_parent->d_inode,
				DN_MODIFY);
		fput(file);
	}
	return ret;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

注意，在调用参数中并不指明在文件中写的位置，因为文件的 file 结构代表着一个上下文，记录着在文件中的“当前位置”。函数 fget()根据打开文件号 fd 找到该已打开文件的 file 结构，这个 inline 函数的代码定义于 include/linux/file.h 中：

==================== fs/file_table.c 125 136 ====================
[sys_write()>fget()]

```
struct file * fget(unsigned int fd)
{
	struct file * file;
	struct files_struct *files = current->files;

	read_lock(&files->file_lock);
	file = fcheck(fd);
	if (file)
		get_file(file);
	read_unlock(&files->file_lock);
	return file;
}
```

这个函数，或者更确切地说是它里面的宏操作 get_file()，一定是与另一个函数 fput()配对使用的，因为这二者一个递增 file 结构中的共享计数，另一个则递减这个计数。哪一个过程在开始时递增了某个 file 结构中的共享计数，就负有责任在结束时递减这个计数。这里 get_file()的定义在 include/linux/fs.h 中：

==================== include/linux/fs.h 521 521 ====================

```
#define get_file(x)	atomic_inc(&(x)->f_count)
```

根据打开文件号找到 file 结构，具体是由 fcheck()完成的，其代码在 file.h 中：

==================== include/linux/file.h 41 52 ====================
[sys_write()>fget()>fcheck()]

```
/*
 * Check whether the specified fd has an open file.
 */
static inline struct file * fcheck(unsigned int fd)
{
	struct file * file = NULL;
	struct files_struct *files = current->files;

	if (fd < files->max_fds)
		file = files->fd[fd];
	return file;
}
```

一个进程要对一个已打开文件进行写操作，应满足几个必要条件。其一是相应 file 结构里 f_mode 字段中的标志位 FMODE_WRITE 为 1。这个字段的内容是在打开文件时根据对系统调用 open()的参数 flags 经过变换而来的，具体见前一节中 filp_open()和 dentry_open()的代码。若标志位 FMODE_WRITE 为 0，则表示这个文件是按“只读”方式打开的，所以该标志位为 1 是写操作的一个必要条件。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

取得了目标文件的 file 结构指针并确认文件是按可写方式打开以后，还要检查文件中从当前位置 f_pos 开始的 count 个字节是否对写操作加上了“强制锁”。这是通过 locks_verify_area()完成的，其代码在 fs.h 中：

==================== include/linux/fs.h 906 913 ====================
[sys_write()>locks_verify_area()]

```
static inline int locks_verify_area(int read_write, struct inode *inode,
                            struct file *filp, loff_t offset,
                            size_t count)
{
    if (inode->i_flock && MANDATORY_LOCK(inode))
        return locks_mandatory_area(read_write, inode, filp, offset, count);
    return 0;
}
```

先检查该文件究竟是否加了锁，以及是否允许使用强制锁。如果确实加了锁，并且可能是强制锁，就进一步通过 locks_mandatory_area()检查所要求的区域是否也被强制锁住了。这个函数的代码在 fs/lock.c 中，我们在这里就不看了。它的算法是很简单的，无非就是扫描该文件的 inode 结构中的 i_flock 队列里面每一个 file_lock 数据结构并进行比对。从这里读者可以看出为什么强制锁并不总是比协调锁优越，因为对每一次读 / 写操作它都要扫描这个队列进行比对，这显然会降低文件读写的速度。特别是如果每次读 / 写的长度都很小，那样花在强制锁检查上的开销所占比例就相当大了。

通过了对强制锁的检查以后，就是写操作本身了。可想而知，不同的文件系统有不同的写操作，具体的文件系统通过其 file_operations 数据结构提供用于写操作的函数指针，就 Ext2 文件系统而言，它有两个这样的数据结构，一个是 ext2_file_operations，另一个是 ext2_dir_operations，视操作的目标为文件或目录而选择其一，在打开该文件时“安装”在其 file 结构中。对于普通的文件，这个函数指针指向 generic_file_write()，其代码在 mm/filemap.c 中，我们分段来看。

==================== mm/filemap.c 2426 2494 ====================
[sys_write()>generic_file_write()]

```
/*
 * Write to a file through the page cache.
 *
 * We currently put everything into the page cache prior to writing it.
 * This is not a problem when writing full pages. With partial pages,
 * however, we first have to read the data into the cache, then
 * dirty the page, and finally schedule it for writing. Alternatively, we
 * could write-through just the portion of data that would go into that
 * page, but that would kill performance for applications that write data
 * line by line, and it's prone to race conditions.
 *
 * Note that this routine doesn't try to keep track of dirty pages. Each
 * file system has to do this all by itself, unfortunately.
 *                                      okir@monad.swb.de
 */
ssize_t
generic_file_write(struct file *file,const char *buf,size_t count,loff_t *ppos)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
    struct inode  *inode = file->f_dentry->d_inode;
    struct address_space *mapping = inode->i_mapping;
    unsigned long limit = current->rlim[RLIMIT_FSIZE].rlim_cur;
    loff_t        pos;
    struct page   *page, *cached_page;
    unsigned long written;
    long      status;
    int       err;

    cached_page = NULL;

    down(&inode->i_sem);

    pos = *ppos;
    err = -EINVAL;
    if (pos < 0)
        goto out;

    err = file->f_error;
    if (err) {
        file->f_error = 0;
        goto out;
    }

    written = 0;

    if (file->f_flags & O_APPEND)
        pos = inode->i_size;

    /*
     * Check whether we've reached the file size limit.
     */
    err = -EFBIG;
    if (limit != RLIM_INFINITY) {
        if (pos >= limit) {
            send_sig(SIGXFSZ, current, 0);
            goto out;
        }
        if (count > limit - pos) {
            send_sig(SIGXFSZ, current, 0);
            count = limit - pos;
        }
    }

    status  = 0;
    if (count) {
        remove_suid(inode);
        inode->i_ctime = inode->i_mtime = CURRENT_TIME;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        mark_inode_dirty_sync(inode);
    }

```

如前所述，inode 结构中有个指针 i_mapping，指向一个 address_space 数据结构，其定义在 include/linux/fs.h 中：

==================== include/linux/fs.h 365 375 ====================

```
struct address_space {
    struct list_head  clean_pages;  /* list of clean pages */
    struct list_head  dirty_pages;  /* list of dirty pages */
    struct list_head  locked_pages; /* list of locked pages */
    unsigned long     nrpages;  /* number of total pages */
    struct address_space_operations *a_ops;   /* methods */
    struct inode      *host;        /* owner: inode, block_device */
    struct vm_area_struct  *i_mmap; /* list of private mappings */
    struct vm_area_struct  *i_mmap_shared; /* list of shared mappings */
    spinlock_t        i_shared_lock;  /* and spinlock protecting it */
};
```

通常这个数据结构就在 inode 结构中，成为 inode 结构的一部分，那就是 i_data（注意切莫与 ext2_inode_info 结构中的数组 i_data[]相混淆）。结构中的队列头 pages 就是用来维持缓冲页面队列的。如果将文件映射到某些进程的用户空间，则指针 i_mmap 指向一串虚存区间，即 vm_area_struct 结构，其中的每一个数据结构都代表着该文件在某一个进程中的空间映射。还有个指针 a_ops 也是很重要的，它指向一个 address_space_operations 数据结构。这个结构中的函数指针给出了缓冲页面与具体文件系统的设备层之间的关系和操作，例如怎样从具体文件系统的设备上读或写一个缓冲页面等等。就 Ext2 文件系统来说，这个数据结构为 ext2_aops，是在 fs/ext2/inode.c 中定义的：

==================== fs/ext2/inode.c 669 676 ====================

```
struct address_space_operations ext2_aops = {
    readpage: ext2_readpage,
    writepage: ext2_writepage,
    sync_page: block_sync_page,
    prepare_write: ext2_prepare_write,
    commit_write: generic_commit_write,
    bmap: ext2_bmap
};
```

我们在系统调用一章中讲过，在某些条件下系统调用会中途流产，而流产以后的对策就是重新执行一遍系统调用。文件操作也是这样。但是，在某些特殊的情况下，如果在中途流产的同时或之前已发生了其它的出错，则此时的重新执行所应该做的只是将出错代码返回给进程，而不应进行任何实质性的操作，file 结构中的 f_error 字段就是为此目的而设的。

如果在打开文件时的参数中将 O_APPEND 标志位设为 1，则表示对此文件的写操作只能在尾端添加，所以要将当前位置 pos 调整到文件的尾端。此外，对每个进程可以使用的各种资源，包括文件大小，是可以加上限制的。进程的 task_struct 结构中有个数组 rlim 就规定了对该进程使用各种资源的上限。其中有一项，即下标为 RLIMIT_FSIZE 处的元素，就表示对该进程的文件大小的限制。如果企图

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

写入的位置超出了这个限制，那就要给这个进程发一个信号 SIGXFSZ，并且让系统调用失败而返回出错代码-EFBIG。

至此，只要待写的长度不为 0，那就是一次有效的写操作了，所以要在 inode 结构中打上时间印记并将该 inode 标志成“脏”，表示其内容应写回设备上的相应索引节点。这里还有一个函数 remove_suid()，其代码在同一文件 mm/filemap.c 中：

==================== mm/filemap.c 2411 2424 ====================

[sys_write()>generic_file_write()>remove_suid()]

```
2411 static inline void remove_suid(struct inode *inode)
2412 {
2413      unsigned int mode;
2414
2415      /* set S_IGID if S_IXGRP is set, and always set S_ISUID */
2416      mode = (inode->i_mode & S_IXGRP)*(S_ISGID/S_IXGRP) | S_ISUID;
2417
2418      /* was any of the uid bits set? */
2419      mode &= inode->i_mode;
2420      if (mode && !capable(CAP_FSETID)) {
2421           inode->i_mode &= ~mode;
2422           mark_inode_dirty(inode);
2423      }
2424 }
```

这段程序的意图恰如其函数名所述。如果当前进程并无设置“ set uid”，即 S_ISUID 标志位的特权，而目标文件的 set uid 标志位 S_ISUID 和 S_ISGID 为 1，则应将 inode 结构中的这些标志位清成 0，也就是剥夺该文件的 set uid 和 set gid 特性。之所以要这样做的原因是简单的（我们把它留给读者，见本段后的附加说明），但是这里的代码却不那么直观。函数中的局部量 mode 实际上是作为屏蔽字使用的，第 2416 行的目的就是注释中所说的。如果 i_mode 中的标志位 S_IXGRP 为 0，那么两项相乘以后的结果也是 0，所以 mode 成为 S_ISUID。而如果 i_mode 中的标志位为 1，那么相乘以后的结果为 S_ISGID，所以 mode 就成为(S_ISGID | S_ISUID)。其余的就比较简单直观了。

此处顺便请读者考虑，如果当前进程不具备设置 S_ISUID 的特权，却具有对一个已经存在的 set uid 可执行文件的写访问权，则它可以把这个文件中的内容全部改写。这样，就相当于当前进程创建了自己的 set uid 可执行文件。

回到 generic_file_write()的代码中继续往下看。

==================== mm/filemap.c 2495 2581 ====================

[sys_write()>generic_file_write()]

```
     while (count) {
          unsigned long bytes, index, offset;
          char *kaddr;
          int deactivate = 1;

          /*
           * Try to find the page in the cache. If it isn't there,
           * allocate a free page.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
		 */
		offset = (pos & (PAGE_CACHE_SIZE -1)); /* Within page */
		index = pos >> PAGE_CACHE_SHIFT;
		bytes = PAGE_CACHE_SIZE - offset;
		if (bytes > count) {
			bytes = count;
			deactivate = 0;
		}

		/*
		 * Bring in the user page that we will copy from _first_.
		 * Otherwise there's a nasty deadlock on copying from the
		 * same page as we're writing to, without it being marked
		 * up-to-date.
		 */
		{ volatile unsigned char dummy;
			__get_user(dummy, buf);
			__get_user(dummy, buf+bytes-1);
		}

		status = -ENOMEM;  /* we'll assign it later anyway */
		page = __grab_cache_page(mapping, index, &cached_page);
		if (!page)
			break;

		/* We have exclusive IO access to the page.. */
		if (!PageLocked(page)) {
			PAGE_BUG(page);
		}

		status = mapping->a_ops->prepare_write(file, page, offset, offset+bytes);
		if (status)
			goto unlock;
		kaddr = page_address(page);
		status = copy_from_user(kaddr+offset, buf, bytes);
		flush_dcache_page(page);
		if (status)
			goto fail_write;
		status = mapping->a_ops->commit_write(file, page, offset, offset+bytes);
		if (!status)
			status = bytes;

		if (status >= 0) {
			written += status;
			count -= status;
			pos += status;
			buf += status;
		}
unlock:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /* Mark it unlocked again and drop the page.. */
        UnlockPage(page);
        if (deactivate)
            deactivate_page(page);
        page_cache_release(page);

        if (status < 0)
            break;
    }
    *ppos = pos;

    if (cached_page)
        page_cache_free(cached_page);

    /* For now, when the user asks for O_SYNC, we'll actually
     * provide O_DSYNC. */
    if ((status >= 0) && (file->f_flags & O_SYNC))
        status = generic_osync_inode(inode, 1); /* 1 means datasync */

    err = written ? written : status;
out:

    up(&inode->i_sem);
    return err;
fail_write:
    status = -EFAULT;
    ClearPageUptodate(page);
    kunmap(page);
    goto unlock;
}
```

写操作的主体部分是由一个 while 循环实现的。循环的次数取决于写的长度和位置，在每一次循环中，只往一个缓冲页面中写，并且将当前位置 pos 相应地向前推进，而剩下未写的长度 count 则逐次减少。首先要根据当前位置 pos 计算出本次循环中要写的缓冲页面 index、在该页面中的起点 offset 以及写入长度 bytes。计算时将整个文件的内容当作一个连续的线性存储空间，将 pos 右移 PAGE_CACHE_SHIFT 位跟将 pos 被页面大小所整除是等价的（但是更快）。计算出了缓冲页面在目标文件中的逻辑序号 index 以后，就通过__grab_cache_page()找到该缓冲页面，如找不到，就分配、建立一个缓冲页面，其代码在 filemap.c 中：

==================== mm/filemap.c 2378 2396 ====================

[sys_write()>generic_file_write()>__grab_cache_page()]

```
static inline struct page * __grab_cache_page(struct address_space *mapping,
                unsigned long index, struct page **cached_page)
{
    struct page *page, **hash = page_hash(mapping, index);
repeat:
    page = __find_lock_page(mapping, index, hash);
    if (!page) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (!*cached_page) {
            *cached_page = page_cache_alloc();
            if (!*cached_page)
                return NULL;
        }
        page = *cached_page;
        if (add_to_page_cache_unique(page, mapping, index, hash))
            goto repeat;
        *cached_page = NULL;
    }
    return page;
}
```

首先是通过杂凑计算从页面杂凑表 page_hash_table 中找到所在或应该在的杂凑队列。与 page_hash()有关的代码和定义在 include/linux/pagemap.h 中：

==================== include/linux/pagemap.h 68 68 ====================

```
#define page_hash(mapping,index) (page_hash_table+_page_hashfn(mapping,index))
```

==================== mm/filemap.c 46 46 ====================

```
extern struct page **page_hash_table;
```

==================== include/linux/pagemap.h 50 66 ====================

```
/*
 * We use a power-of-two hash table to avoid a modulus,
 * and get a reasonable hash by knowing roughly how the
 * inode pointer and indexes are distributed (ie, we
 * roughly know which bits are "significant")
 *
 * For the time being it will work for struct address_space too (most of
 * them sitting inside the inodes). We might want to change it later.
 */
extern inline unsigned long _page_hashfn(struct address_space * mapping, unsigned long index)
{
#define i (((unsigned long) mapping)/(sizeof(struct inode) & ~ (sizeof(struct inode) - 1)))
#define s(x) ((x)+((x)>>PAGE_HASH_BITS))
    return s(i+index) & (PAGE_HASH_SIZE-1);
#undef i
#undef s
}
```

值得注意的是，在杂凑计算中除页面的逻辑序号 index 外还使用了指针 mapping，这是因为页面在文件中的逻辑序号在系统范围内并不是惟一的。

这里 page_hash()返回的是一个指向数组 page_hash_table 中某一元素的指针，而这个元素本身则又是一个 page 结构指针，指向队列中的第一个 page 结构。

找到了目标页面所在，或者应该在的杂凑队列后，就要搜索这个队列，找到该页面的 page 结构，这是由__flnd_lock_page()完成的。我们在这里就不看这些低层函数的代码了，读者不妨回顾一下第 2 章中的代码。

总之，如果在队列中找到了目标页面就万事大吉，找不到就要通过 page_cache_alloc()分配一个空

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

闲（并且空白）的页面，并通过 add_to_page_cache_unique()将其链入相应的杂凑队列中。不过，在调用__grab_cache_page()时也可以通过调用参数带下一个空闲页面来，此时就把带下来的页面先用掉，而不分配新的页面了。

这样，只要系统中还有可用的页面，从__grab_cache_page()返回到 generic_file_write()中时一定已经有了一个缓冲页面，只是这个页面有可能是个新分配的空白页面。新分配的空白页面与业已存在的缓冲页面除在内容上有根本性的区别外，在结构上也有个重要的区别。那就是前面所讲的，缓冲页面一方面与一个 page 结构相联系，另一方面又要与若干记录块缓冲区的头部，即 buffer_head 数据结构相联系，而新分配的页面则尚无 buffer_head 结构与之挂钩。所以，对于新分配的空白页面一来要为其配备相应的 buffer_head 数据结构，二来要将目标页面的内容先从设备中读入（因为写操作未必是整个页面的写入）。不仅如此，就是业已存在的老页面也有个缓冲页面中的内容是否“ up_to_date”，即是否一致的问题。这里所谓“一致”，是指缓冲页面或缓冲区中的内容与设备上的逻辑内容（不一定是物理内容）一致，详细情况可看后面对_block_commit_write()的讨论。换言之，在开始写入前还要做一些准备工作，而这些准备工作与具体文件系统有关，所以由具体的 address_space_operations 数据结构通过函数指针 prepare_write 提供具体的操作函数，就 Ext2 文件系统而言，这个函数为 ext2_prepare_write()，其代码在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 661 664 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()]

```
static int ext2_prepare_write(struct file *file, struct page *page, unsigned from, unsigned to)
{
    return block_prepare_write(page,from,to,ext2_get_block);
}
```

这里 block_prepare_write()是个通用的函数，定义于 fs/buffer.c，其具体的低层操作由作为参数传递的函数指针决定，而这里传下去的函数为 ext2_get_block()。

==================== fs/buffer.c 1832 1842 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()]

```
int block_prepare_write(struct page *page, unsigned from, unsigned to,
            get_block_t *get_block)
{
    struct inode *inode = page->mapping->host;
    int err = __block_prepare_write(inode, page, from, to, get_block);
    if (err) {
        ClearPageUptodate(page);
        kunmap(page);
    }
    return err;
}
```

显然，这个函数的主体是__block_prepare_write()，它的代码也在同一文件 fs/buffer.c 中：

==================== fs/buffer.c 1557 1625 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()]

```
static int __block_prepare_write(struct inode *inode, struct page *page,
```

PDF-XChange
Click to buy NOW!
www.docu-track.com

```
        unsigned from, unsigned to, get_block_t *get_block)
{
    unsigned block_start, block_end;
    unsigned long block;
    int err = 0;
    unsigned blocksize, bbits;
    struct buffer_head *bh, *head, *wait[2], **wait_bh=wait;
    char *kaddr = kmap(page);

    blocksize = inode->i_sb->s_blocksize;
    if (!page->buffers)
        create_empty_buffers(page, inode->i_dev, blocksize);
    head = page->buffers;

    bbits = inode->i_sb->s_blocksize_bits;
    block = page->index << (PAGE_CACHE_SHIFT - bbits);

    for(bh = head, block_start = 0; bh != head || !block_start;
       block++, block_start=block_end, bh = bh->b_this_page) {
        if (!bh)
            BUG();
        block_end = block_start+blocksize;
        if (block_end <= from)
            continue;
        if (block_start >= to)
            break;
        if (!buffer_mapped(bh)) {
            err = get_block(inode, block, bh, 1);
            if (err)
                goto out;
            if (buffer_new(bh)) {
                unmap_underlying_metadata(bh);
                if (Page_Uptodate(page)) {
                    set_bit(BH_Uptodate, &bh->b_state);
                    continue;
                }
                if (block_end > to)
                    memset(kaddr+to, 0, block_end-to);
                if (block_start < from)
                    memset(kaddr+block_start, 0, from-block_start);
                if (block_end > to || block_start < from)
                    flush_dcache_page(page);
                continue;
            }
        }
        if (Page_Uptodate(page)) {
            set_bit(BH_Uptodate, &bh->b_state);
            continue;
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            if (!buffer_uptodate(bh) &&
                (block_start < from || block_end > to)) {
                ll_rw_block(READ, 1, &bh);
                *wait_bh++=bh;
            }
        }
        /*
         * If we issued read requests - let them complete.
         */
        while(wait_bh > wait) {
            wait_on_buffer(*--wait_bh);
            err = -EIO;
            if (!buffer_uptodate(*wait_bh))
                goto out;
        }
        return 0;
out:
        return err;
}
```

参数 get_block 是个函数指针，对于 Ext2 文件系统它指向 ext2_get_block()。这个函数的作用是为一个给定的缓冲页面中的记录块缓冲区做好写入的准备。如前所述，因具体文件系统和设备的不同，记录块的大小也可能不同，其实际的大小记录在设备的超级块中，从而在 super_block 结构中。一个页面由若干个记录块构成。对于原已存在的页面，这些缓冲区的 buffer_head 结构都通过指针 b_this_page 指向同一页面中的下一个 buffer_head，向形成缓冲页面 page 结构里的队列 buffers。而如果是新分配建立的页面，则要通过 create_empty_buffers()为该页面配备好相应的 buffer_head 结构，并建立起这个队列。这个函数的代码也在 buffer.c 中：

==================== fs/buffer.c 1426 1445 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>create_empty_buffers()]

```
static void create_empty_buffers(struct page *page, kdev_t dev, unsigned long blocksize)
{
        struct buffer_head *bh, *head, *tail;

        head = create_buffers(page, blocksize, 1);
        if (page->buffers)
            BUG();

        bh = head;
        do {
            bh->b_dev = dev;
            bh->b_blocknr = 0;
            bh->b_end_io = NULL;
            tail = bh;
            bh = bh->b_this_page;
        } while (bh);
        tail->b_this_page = head;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1443     page->buffers = head;
1444     page_cache_get(page);
1445 }
```

这里的 page_cache_get()只是递增 page 结构中的共享计数。

回到__block_prepare_write()的代码中。如前所述，虽然在文件系统层次上是以页面为单位缓冲的，在设备层次上却是以记录块为单位缓冲的。所以，如果一个缓冲页面的内容是一致的，就意味着构成这个页面的所有记录块的内容都一致，反过来，如果一个缓冲页面不一致，则未必每个记录块都不一致。因此，要根据写入的位置和长度找到具体涉及的记录块，针对这些记录块做写入的准备。

做些什么准备呢？简而言之就是使有关记录块缓冲区的内容与设备上相关记录块的内容相一致。如果缓冲页面已经建立起对物理记录块的映射，则需要做的只是检查一下目录记录块的内容是否一致（见第 1607 行和 1608 行），如果不一致就通过 ll_rw_block()将设备上的记录块读到缓冲区中。由此可见，对文件的写操作实际上往往是“写中有读”、“欲写先读”。

可是，如果缓冲页面是新的，尚未映射到物理记录块呢？那就比较复杂一些了，因为根据页面号、页面大小、记录块大小计算所得的记录块号（见 1585 行）只是文件内部的逻辑块号，这是在假定文件的内容为连续的线性空间这么一个前提下计算出来的，而实际的记录块在设备上的位置则是动态地分配和回收的。另一方面，在设备层也根本没有文件的概念，而只能按设备上的记录块号读写。设备上的记录块号也是逻辑块号，与设备上的记录块位图相对应。而设备上的逻辑块号与物理记录块有着一一对应的关系，所以在文件层也可以认为是“物理块号”。总而言之，这里有一个从文件内的逻辑记录块号到设备上的记录块号之间的映射问题。缺少了对这种映射关系的描述，就无法根据文件内的逻辑块号在设备上找到相应的记录块。可想而知，不同的文件系统可能有不同的映射关系或过程，这就是要由作为参数传给__block_prepare_write()的函数指针 get_block 来完成这种映射的原因。对于 Ext2 文件系统这个函数是 ext2_get_block()，在 fs/ext2/inode.c

==================== fs/ext2/inode.c 506 575 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()]

```
506 static int ext2_get_block(struct inode *inode, long iblock, struct buffer_head *bh_result, int create)
507 {
508     int err = -EIO;
509     int offsets[4];
510     Indirect chain[4];
511     Indirect *partial;
512     unsigned long goal;
513     int left;
514     int depth = ext2_block_to_path(inode, iblock, offsets);
515
516     if (depth == 0)
517         goto out;
518
519     lock_kernel();
520 reread:
521     partial = ext2_get_branch(inode, depth, offsets, chain, &err);
522
523     /* Simplest case - block found, no allocation needed */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (!partial) {
got_it:
        bh_result->b_dev = inode->i_dev;
        bh_result->b_blocknr = le32_to_cpu(chain[depth-1].key);
        bh_result->b_state |= (1UL << BH_Mapped);
        /* Clean up and exit */
        partial = chain+depth-1; /* the whole chain */
        goto cleanup;
    }

    /* Next simple case - plain lookup or failed read of indirect block */
    if (!create || err == -EIO) {
cleanup:
        while (partial > chain) {
            brelse(partial->bh);
            partial--;
        }
        unlock_kernel();
out:
        return err;
    }

    /*
     * Indirect block might be removed by truncate while we were
     * reading it. Handling of that case (forget what we've got and
     * reread) is taken out of the main path.
     */
    if (err == -EAGAIN)
        goto changed;

    if (ext2_find_goal(inode, iblock, chain, partial, &goal) < 0)
        goto changed;

    left = (chain + depth) - partial;
    err = ext2_alloc_branch(inode, left, goal,
                       offsets+(partial-chain), partial);
    if (err)
        goto cleanup;

    if (ext2_splice_branch(inode, iblock, chain, partial, left) < 0)
        goto changed;

    bh_result->b_state |= (1UL << BH_New);
    goto got_it;

changed:
    while (partial > chain) {
        bforget(partial->bh);
        partial--;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }
        goto reread;
    }
```

参数 iblock 表示所处理的记录块在文件中的逻辑块号，inode 则指向文件的 inode 结构；参数 create 表示是否需要创建。从__block_prepare_write()中传下的实际参数值为 1，所以我们在这里只关心 create 为 1 的情景。从文件内块号到设备上块号的映射，最简单最迅速的当然莫过于使用一个以文件内块号为下标的线性数组，并且将这个数组置于索引节点 inode 结构中。可是，那样就需要很大的数组，从而使索引节点和 inode 结构也变得很大，或者就得使用可变长度的索引节点而使文件系统的结构更加复杂。

另一种方法是采用间接寻址，也就是将上述的数组分块放在设备上本来可用于存储数据的若干记录块中，而将这些记录块的块号放在索引节点和 inode 结构中。这些记录块虽然在设备上的数据区（而不是索引节点区）中，却并不构成文件本身的内容，而只是一些管理信息。由于索引节点（和 inode 结构）应该是固定大小的，所以当文件较大时还要将这种间接寻址的结构框架做成树状或链状，这样才能随着文件本身的大小而扩展其容量，显然，这种方法解决了容量的问题，但是降低了运行时的效率。

基于这些考虑，从 Unix 早期就采用了一种折衷的方法，可以说是直接与间接相结合。其方法是把整个文件的记录块寻址分成几个部分来实现。第一部分是个以文件内块号为下标的数组，这是采用直接映射的部分，对于较小的文件这一部分就够用了。由于根据文件内块号就可以在 inode 结构里的数组中直接找到相应的设备上块号，所以效率很高。至于比较大的文件，其开头那一部分记录块号也同样直接就可以找到，但是当文件的大小超出这一部分的容量时，超出的那一部分就要采用间接寻址了。Ext2 文件系统的这一部分的大小为 12 个记录块，即数组的大小为 12。当记录块大小为 1K 字节时，相应的文件大小为 12K 字节。在 Ext2 文件系统的 ext2_inode_info 结构中，有个大小为 15 的整型数组 i_data[]，其开头 12 个元素即用于此项目。当文件大小超过这一部分的容量时，该数组中的第 13 个元素指向一个记录块，这个记录块的内容也是一个整型数组，其中的每个元素都指向一个设备上记录块。如果记录块大小为 1K 字节，则该数组的大小为 256，也就是说间接寻址的容量为 256 个记录块，即 256K 字节。这样，两个部分的总容量为 12K + 256K = 268K 字节。可是，更大的文件还是容纳不下，所以超过此容量的部分要进一步采用双重（二层）间接寻址。此时 inode 结构里 i_data[]数组中的第 14 个元素指向另一个记录块，该记录块的内容也是一个数组，但是每个元素都指向另一个记录块中的数组，那才是文件内块号至设备上块号的映射表。这么一来，双重间接寻址部分的能力为 256 × 256 = 64K 个记录块，即 64M 字节。依此类推，数组 i_data[]中的第 15 个元素用于三重（三层）间接寻址，这一部分的容量可达 256 × 256 × 256 = 16M 个记录块，也就是 16G 字节，所以，对于 32 位结构的系统，当记录块大小为 1K 字节时，文件的最大容量为 16G + 64M + 256K + 12K。如果设备的容量大于这个数值，就得采用更大的记录块大小了。图 5.7 是一个关于直接和间接映射的示意图。

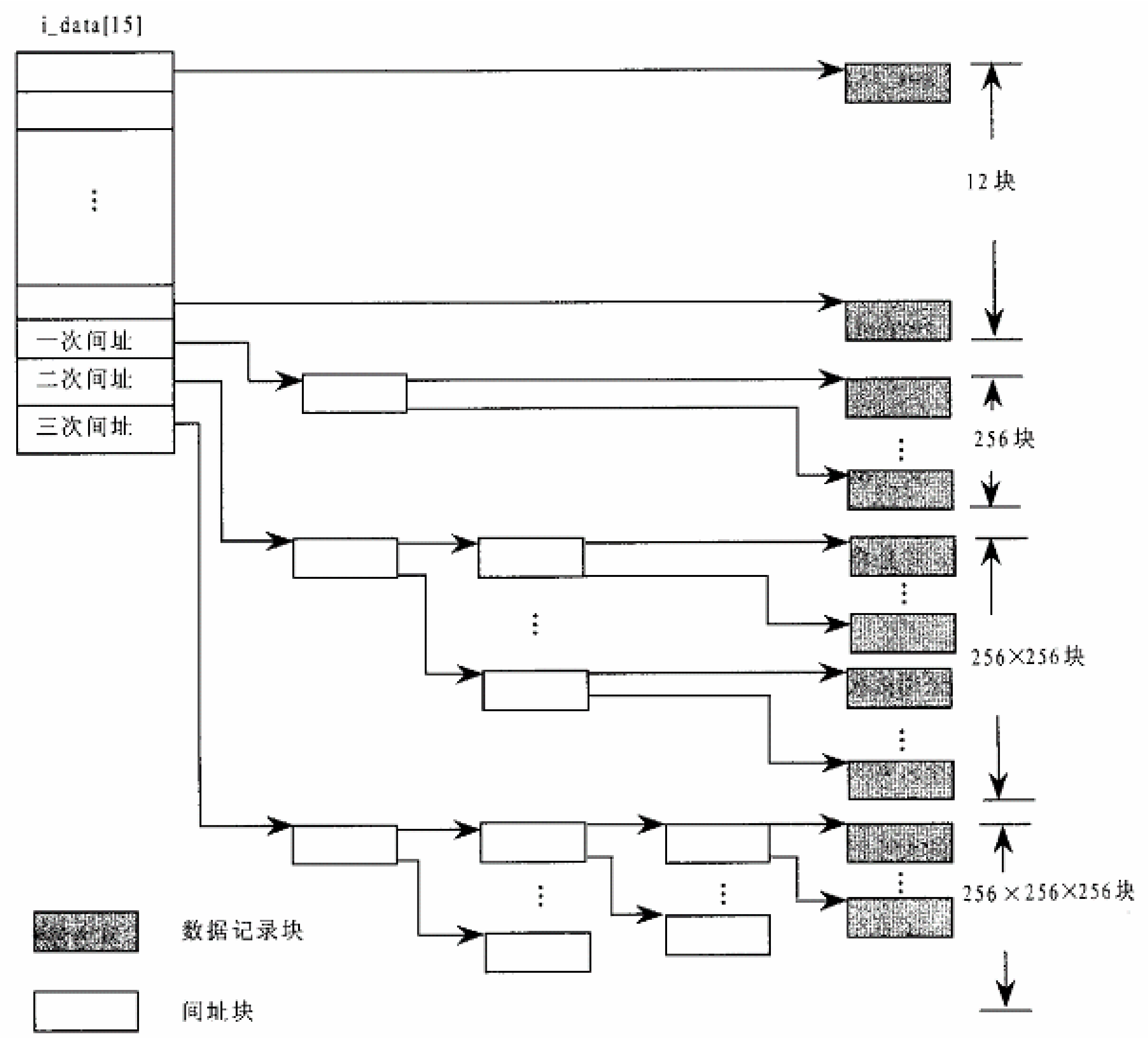


图 5.7　多重间接映射示意图

从严格意义上说，i_data[]其实不能说是一个数组，因为它的元素并不都是同一类型的。但是，从另一个角度说，则这些元素毕竟都是长整数，都代表着设备上一个记录块，只是这些记录块的用途不同而已。

这里还要注意，在 inode 结构中有个成分名为 i_data，这是一个 address_space 数据结构。而作为 inode 结构一部分的 ext2_inode_info 结构中，也有个名为 i_data 的数组，实际上就是记录块映射表，二者毫无关系。从概念上说，inode 结构是设备上的索引节点即 ext2_inode 结构的对应物，但实际上 inode 结构中的很多内容并非来自 ext2_inode 结构。相比之下，ext2_inode_info 结构中的信息才是基本上与设备上的索引节点相对应的。例如，与 ext2_inode_info 中的数组 i_data[]相对应，在 ext2_inode 结构中也有个数组 i_block[]，两个数组的大小也相同。而 ext2_inode_info 中的数组 i_data[]之所以不能再大一些，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

就是因为索引节点中的数组 i_block[]只能这么大了。那么内存中的 inode 结构为什么与设备上的索引节点有相当大的不同呢？原因在于设备上索引节点的大小受到更多的限制，所以在索引节点中只能存储必需的信息，而且是相对静态的信息。而内存中的 inode 结构就不同了，它受的限制比较小，除了来自索引节点的必需信息外还可以用来保存一些为方便和提高运行效率所需的信息，还有一些运行时需要的更为动态的信息，如各种指针，以及为实现某些功能所需的信息，如 i_sock，i_pipe，i_wait 和 i_flock 等等。还应提醒读者，设备上的索引节点数量与设备的大小以及文件系统格式的设计有直接的关系，设备上的每一个文件都有一个索引节点，但是内存中的 inode 结构则主要是缓冲性质的，实际上只有很小一部分文件在内存中建立并保持 inode 结构。

有了这些背景知识，我们就可以深入到 ext2_get_block()的代码中了。这里用到的一些宏定义都在 include/linux/ext2_fs.h 中：

==================== include/linux/ext2_fs.h 85 85 ====================

```
# define EXT2_BLOCK_SIZE(s)      ((s)->s_blocksize)
```

==================== include/linux/ext2_fs.h 90 90 ====================

```
#define  EXT2_ADDR_PER_BLOCK(s)      (EXT2_BLOCK_SIZE(s) / sizeof (__u32))
```

==================== include/linux/ext2_fs.h 97 97 ====================

```
#define  EXT2_ADDR_PER_BLOCK_BITS(s) ((s)->u.ext2_sb.s_addr_per_block_bits)
```

==================== include/linux/ext2_fs.h 174 181 ====================

```
/*
 * Constants relative to the data blocks
 */
#define  EXT2_NDIR_BLOCKS       12
#define  EXT2_IND_BLOCK              EXT2_NDIR_BLOCKS
#define  EXT2_DIND_BLOCK             (EXT2_IND_BLOCK + 1)
#define  EXT2_TIND_BLOCK             (EXT2_DIND_BLOCK + 1)
#define  EXT2_N_BLOCKS          (EXT2_TIND_BLOCK + 1)
```

这些定义中的 EXT2_NDIR_BLOCKS 为 12，表示直接映射的记录块数量。EXT2_IND_BLOCK 的值也是 12，表示在 i_data[]数组中用于一次间接映射的元素下标。而 EXT2_DIND_BLOCK 和 EXT2_TIND_BLOCK 则分别为用于二次间接和三次间接的元素下标。至于 EXT2_N_BLOCKS 则为 i_data[]数组的大小。

首先根据文件内块号计算出这个记录块落在哪一个区间，要采用几重映射（1 表示直接）。这是由 ext2_block_to_path()完成的，其代码在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 144 202 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_block_to_path()]

```
/**
 *  ext2_block_to_path - parse the block number into array of offsets
 *  @inode: inode in question (we are only interested in its superblock)
 *  @i_block: block number to be parsed
 *  @offsets: array to store the offsets in
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   To store the locations of file's data ext2 uses a data structure common
 *   for UNIX filesystems - tree of pointers anchored in the inode, with
 *   data blocks at leaves and indirect blocks in intermediate nodes.
 *   This function translates the block number into path in that tree -
 *   return value is the path length and @offsets[n] is the offset of
 *   pointer to (n+1)th node in the nth one. If @block is out of range
 *   (negative or too large) warning is printed and zero returned.
 *
 *   Note: function doesn't find node addresses, so no IO is needed. All
 *   we need to know is the capacity of indirect blocks (taken from the
 *   inode->i_sb).
 */

/*
 * Portability note: the last comparison (check that we fit into triple
 * indirect block) is spelled differently, because otherwise on an
 * architecture with 32-bit longs and 8Kb pages we might get into trouble
 * if our filesystem had 8Kb blocks. We might use long long, but that would
 * kill us on x86. Oh, well, at least the sign propagation does not matter -
 * i_block would have to be negative in the very beginning, so we would not
 * get there at all.
 */

static int ext2_block_to_path(struct inode *inode, long i_block, int offsets[4])
{
     int ptrs = EXT2_ADDR_PER_BLOCK(inode->i_sb);
     int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(inode->i_sb);
     const long direct_blocks = EXT2_NDIR_BLOCKS,
          indirect_blocks = ptrs,
          double_blocks = (1 << (ptrs_bits * 2));
     int n = 0;

     if (i_block < 0) {
          ext2_warning (inode->i_sb, "ext2_block_to_path", "block < 0");
     } else if (i_block < direct_blocks) {
          offsets[n++] = i_block;
     } else if ( (i_block -= direct_blocks) < indirect_blocks) {
          offsets[n++] = EXT2_IND_BLOCK;
          offsets[n++] = i_block;
     } else if ((i_block -= indirect_blocks) < double_blocks) {
          offsets[n++] = EXT2_DIND_BLOCK;
          offsets[n++] = i_block >> ptrs_bits;
          offsets[n++] = i_block & (ptrs - 1);
     } else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
          offsets[n++] = EXT2_TIND_BLOCK;
          offsets[n++] = i_block >> (ptrs_bits * 2);
          offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
          offsets[n++] = i_block & (ptrs - 1);
     } else {
```

```
        ext2_warning (inode->i_sb, "ext2_block_to_path", "block > big");
    }
    return n;
}
```

根据上面这些宏定义，在记录块大小为 1K 字节时，代码中的局部量 ptrs 赋值为 256，从而 indirect_blocks 也是 256。与 ptrs 相对应的 ptrs_bits 则为 8，因为 256 是由 1 左移 8 位而成的。同样地，二次间接的容量 double_blocks 就是由 1 左移 16 位，即 64K。而三次间接的容量为由 1 左移 24 位，即 16M。

除映射“深度”外，还要算出在每一层映射中使用的位移量，即数组中的下标，并将计算的结果放在一个数组 offset[]中备用。例如，文件内块号 10 不需要间接映射，一步就能到位，所以返回值为 1，并于 offset[0]中返回在第一个数组，即 i_data[]中的位移 10。可是，假若文件内块号为 20，则返回值为 2，而 offset[0]为 12，offset[1]为 8。这样，就在数组 offset[]中为各层映射提供了一条路线。数组的大小是 4，因为最多就是三重间接。参数 offset 实际上是一个指针，在 C 语言里数组名与指针是等价的。

如果 ext2_block_to_path()的返回值为 0 表示出了错，因为文件内块号与设备上块号之间至少也得映射一次。出错的原因可能是文件内块号太大或为负值，或是下面要讲到的冲突。否则，就进一步从磁盘上逐层读入用于间接映射的记录块，这是由 ext2_get_branch()完成的。

==================== fs/ext2/inode.c 204 270 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_get_branch()]

```
/**
 *  ext2_get_branch - read the chain of indirect blocks leading to data
 *  @inode: inode in question
 *  @depth: depth of the chain (1 - direct pointer, etc.)
 *  @offsets: offsets of pointers in inode/indirect blocks
 *  @chain: place to store the result
 *  @err: here we store the error value
 *
 *  Function fills the array of triples <key, p, bh> and returns %NULL
 *  if everything went OK or the pointer to the last filled triple
 *  (incomplete one) otherwise. Upon the return chain[i].key contains
 *  the number of (i+1)-th block in the chain (as it is stored in memory,
 *  i.e. little-endian 32-bit), chain[i].p contains the address of that
 *  number (it points into struct inode for i==0 and into the bh->b_data
 *  for i>0) and chain[i].bh points to the buffer_head of i-th indirect
 *  block for i>0 and NULL for i==0. In other words, it holds the block
 *  numbers of the chain, addresses they were taken from (and where we can
 *  verify that chain did not change) and buffer_heads hosting these
 *  numbers.
 *
 *  Function stops when it stumbles upon zero pointer (absent block)
 *      (pointer to last triple returned, *@err == 0)
 *  or when it gets an IO error reading an indirect block
 *      (ditto, *@err == -EIO)
 *  or when it notices that chain had been changed while it was reading
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *        (ditto, *@err == -EAGAIN)
 *   or when it reads all @depth-1 indirect blocks successfully and finds
 *   the whole chain, all way to the data (returns %NULL, *err == 0).
 */
static inline Indirect *ext2_get_branch(struct inode *inode,
                        int depth,
                        int *offsets,
                        Indirect chain[4],
                        int *err)
{
    kdev_t dev = inode->i_dev;
    int size = inode->i_sb->s_blocksize;
    Indirect *p = chain;
    struct buffer_head *bh;

    *err = 0;
    /* i_data is not going away, no lock needed */
    add_chain (chain, NULL, inode->u.ext2_i.i_data + *offsets);
    if (!p->key)
        goto no_block;
    while (--depth) {
        bh = bread(dev, le32_to_cpu(p->key), size);
        if (!bh)
            goto failure;
        /* Reader: pointers */
        if (!verify_chain(chain, p))
            goto changed;
        add_chain(++p, bh, (u32*)bh->b_data + *++offsets);
        /* Reader: end */
        if (!p->key)
            goto no_block;
    }
    return NULL;

changed:
    *err = -EAGAIN;
    goto no_block;
failure:
    *err = -EIO;
no_block:
    return p;
}
```

与前一个函数中的 offset[]一样，这里的参数 chain[]也是一个指针，指向一个 Indirect 结构数组，其类型定义于 fs/ext2/inode.c：

==================== fs/ext2/inode.c 125 129 ====================

```
typedef struct {
    u32  *p;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    u32  key;
    struct buffer_head *bh;
} Indirect;
```

根据数组 offset[]（参数 offsets 指向这个数组）的指引，这个函数逐层将用于记录块号映射的记录块读入内存，并将指向缓冲区的指针保存在数组 chain[]的相应元素，即 Indirect 结构中。同时，还要使该 Indirect 结构中的指针 p 指向本层记录块号映射表（数组）中的相应表项，并使字段 key 持有该表项的内容。具体 Indirect 结构的内容是由 add_chain()设置的：

==================== fs/ext2/inode.c 131 135 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_get_branch()>add_chain()]

```
static inline void add_chain(Indirect *p, struct buffer_head *bh, u32 *v)
{
    p->key = *(p->p = v);
    p->bh = bh;
}
```

仍以前面所举的两个逻辑块为例。文件内块号 10 不需要间接映射，所以只用 chain[0]一个 Indirect 结构。其指针 bh 为 NULL，因为没有用于间接映射的记录块；指针 p 指向映射表中直接映射部分下标为 10 处，即&inode->u.ext2_i.i_data[10]；而 key 则持有该表项的内容，即所映射的设备上块号。相比之下，文件内块号 20 需要一次间接映射，所以要用 chain[0]和 chain[1]两个表项。第一个表项 chain[0]中的指针 bh 仍为 NULL，因为在这一层上没有用于间接映射的记录块；指针 p 指向映射表中下标为 12 处，即&inode->u.ext2_i.i_data[12]，这是用于一层间接映射的表项；而 key 则持有该表项的内容，即用于一层间接映射的记录块的设备上块号。第二个表项 chain[1]中的指针 bh 则指向该记录块的缓冲区，这个缓冲区的内容就是用作映射表的一个整数数组。所以 chain[1]中的指针 p 指向这个数组中下标为 8 处，而 key 则持有该表项的内容，即经过间接映射后的设备上块号。这样，根据具体映射的深度 depth，数组 chain[]中的最后一个元素，更确切地说是 chain[depth-1].key，总是持有目标记录块的物理块号。而从 chain[]中的第一个元素 chain[0]到具体映射的最后一个元素 chain[depth-1]，则提供了具体映射的整个路径，构成了一条映射链，这也是数组名 chain 的由来。如果把映射的过程看成“爬树”的过程，则一条映射链也可看成决定着树上的一个分枝，所以叫 ext2_get_branch()。

给定 chain[]数组中的两个 Indirect 结构，可以通过一个函数 verify_chain()检查它们是否构成一条有效的映射链（fs/ext2/inode.c）：

==================== fs/ext2/inode.c 137 142 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_get_branch()>verify_chain()]

```
static inline int verify_chain(Indirect *from, Indirect *to)
{
    while (from <= to && from->key == *from->p)
        from++;
    return (from > to);
}
```

在 ext2_get_branch()的代码中可以看到：从设备上逐层读入用于间接映射的记录块时，每通过 bread()

PDF-XCHANGE Click to buy NOW! www.docu-track.com

读入一个记录块以后都要调用 verify_chain()再检查一下映射链的有效性，实质上是检查各层映射表中有关的内容是否改变了（见代码中的条件 from->key==*from->p）。为什么有可能改变呢？这是因为从设备上读入一个记录块是费时间的操作，当前进程会进入睡眠而系统会调度其它进程运行。这样，就有可能发生冲突了。例如，被调度运行的进程可能会打开这个文件并加以截尾，即把文件原有的内容删除。所以，当因等待读入中间记录块而进入睡眠的进程恢复运行的时候，可能会发现原来有效的映射链已经变成无效了，此时 ext2_get_branch()返回一个出错代码-EAGAIN。当然，发生这种情况的概率是很小的，但是一个软件是否“健壮”就在于是否考虑到了所有的可能。至于 bread()，那已是属于设备驱动的范畴，读者可参阅块设备驱动一章中的有关内容。

这样，ext2_get_branch()深化了 ext2_block_to_path()所取得的结果，二者合在一起基本完成了从文件内块号到设备上块号的映射。

从 ext2_get_branch()返回的值有两种可能。首先，如果顺利完成了映射则返回值为 NULL。其次，如果在某一层上发现映射表内的相应表项为 0，则说明这个表项（记录块）原来并不存在，现在因为写操作而需要扩充文件的大小。此时返回指向该层 Indirect 结构的指针，表示映射在此“断裂”了。此外，如果映射的过程中出了错，例如读记录块失败，则通过参数 err 返回一个出错代码。

回到 ext2_get_block()的代码中。如果顺利完成了映射，就把所得的结果填入作为参数传下来的缓冲区结构 bh_result 中，然后把映射过程中读入的缓冲区（用于间接映射）全都释放，就最后完成了记录块号的映射。

可是，要是 ext2_get_branch()返回了一个非 0 指针（代码中的局部量 partial），那就说明映射在某一层上断裂了。根据映射的深度和断裂的位置（层次），这个记录块也许还只是个中间的、用于间接映射的记录块，也许就是最终的目标记录块。总之，在这种情况下，要在设备上为目标记录块以及可能需要的中间记录块分配空间。

首先从本文件的角度为目标记录块的分配提出一个“建议块号”，由 ext2_find_goal()确定（fs/ext2/inode.c）：

==================== fs/ext2/inode.c 309 348 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_find_goal()]

```
/**
 *  ext2_find_goal - find a prefered place for allocation.
 *  @inode: owner
 *  @block:  block we want
 *  @chain:  chain of indirect blocks
 *  @partial: pointer to the last triple within a chain
 *  @goal:    place to store the result.
 *
 *  Normally this function find the prefered place for block allocation,
 *  stores it in *@goal and returns zero. If the branch had been changed
 *  under us we return -EAGAIN.
 */

static inline int ext2_find_goal(struct inode *inode,
                     long block,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                     Indirect chain[4],
                     Indirect *partial,
                     unsigned long *goal)
{
    /* Writer: ->i_next_alloc* */
    if (block == inode->u.ext2_i.i_next_alloc_block + 1) {
        inode->u.ext2_i.i_next_alloc_block++;
        inode->u.ext2_i.i_next_alloc_goal++;
    }
    /* Writer: end */
    /* Reader: pointers, ->i_next_alloc* */
    if (verify_chain(chain, partial)) {
        /*
         * try the heuristic for sequential allocation,
         * failing that at least try to get decent locality.
         */
        if (block == inode->u.ext2_i.i_next_alloc_block)
            *goal = inode->u.ext2_i.i_next_alloc_goal;
        if (!*goal)
            *goal = ext2_find_near(inode, partial);
        return 0;
    }
    /* Reader: end */
    return -EAGAIN;
}
```

参数 block 为文件内逻辑块号，goal 则用来返回所建议的设备上目标块号。从本文件的角度，当然希望所有的记录块在设备上都紧挨在一起并且连续。为此目的，在 ext2_inode_info 数据结构中设置了两个字段，即 i_next_alloc_block 和 i_next_alloc_goal。前者用来记录下一次要分配的文件内块号，后者则用来记录希望下一次能分配的设备上块号。在正常的情况下对文件的扩充是顺序的，所以每次的文件内块号都与前一次的连续，而理想的设备上块号也同样连续，二者平行地向前推进。当然，这只是从一个特定文件的角度提出的建议值，能否实现还要看条件是否允许，但是内核会尽量满足要求，不能满足也会尽可能靠近建议的块号分配。

可是，文件内逻辑块号也有可能不连续，也就是说对文件的扩充是跳跃的，新的逻辑块号与文件原有的最后一个逻辑块号之间留一下了“空洞”。这种情况发生在通过系统调用 lseek()将已打开文件的当前读写位置推进到了超出文件末尾之后，可以在文件中造成这样的空洞是 lseek()的一个重要性质。在这种情况下怎样确定对设备上记录块号的建议值呢？这就是调用 ext2_find_near()的目的。

==================== fs/ext2/inode.c 272 307 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_find_goal()>ext2_find_near()]

```
/**
 *  ext2_find_near - find a place for allocation with sufficient locality
 *  @inode: owner
 *  @ind: descriptor of indirect block.
 *
```

```
 *   This function returns the prefered place for block allocation.
 *   It is used when heuristic for sequential allocation fails.
 *   Rules are:
 *     + if there is a block to the left of our position - allocate near it.
 *     + if pointer will live in indirect block - allocate near that block.
 *     + if pointer will live in inode - allocate in the same cylinder group.
 *   Caller must make sure that @ind is valid and will stay that way.
 */

static inline unsigned long ext2_find_near(struct inode *inode, Indirect *ind)
{
	u32 *start = ind->bh ? (u32*) ind->bh->b_data : inode->u.ext2_i.i_data;
	u32 *p;

	/* Try to find previous block */
	for (p = ind->p - 1; p >= start; p--)
		if (*p)
			return le32_to_cpu(*p);

	/* No such thing, so let's try location of indirect block */
	if (ind->bh)
		return ind->bh->b_blocknr;

	/*
	 * It is going to be refered from inode itself? OK, just put it into
	 * the same cylinder group then.
	 */
	return (inode->u.ext2_i.i_block_group *
		EXT2_BLOCKS_PER_GROUP(inode->i_sb)) +
		 le32_to_cpu(inode->i_sb->u.ext2_sb.s_es->s_first_data_block);
}
```

首先将起点 start 设置成指向当前映射表（映射过程中首次发现映射断裂的那个映射表）的起点，然后在当前映射表内往回搜索。如果要分配的是空洞后面的第一个记录块，那就要往回找到空洞之前的表项所对应的物理块号，并以此为建议块号。当然，这个物理块已经在使用中，这个要求是不可能满足的。但是，内核在分配物理记录块时会在位图中从这里开始往前搜索，就近分配空闲的物理记录块。还有一种可能，就是空洞在一个间接映射表的开头处，所以往回搜索时在本映射表中找不到空洞之前的表项，此时就以间接映射表本身所在的记录块作为建议块号。同样，内核在分配物理块号时也会从此开始向前搜索。最后还有一种可能，空洞就在文件的开头处，那就以索引节点所在块组的第一个数据记录块作为建议块号。

回到 ext2_get_block()的代码中。设备上具体记录块的分配，包括目标记录块和可能需要的用于间接映射的中间记录块，以及映射的建立，是由 ext2_alloc_branch()完成的。调用之前先要算出映射断裂点离终点的距离，也就是还有几层映射需要建立。有关的代码都在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 350 423 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

>ext2_alloc_branch()]

```
/**
 *   ext2_alloc_branch - allocate and set up a chain of blocks.
 *   @inode: owner
 *   @num: depth of the chain (number of blocks to allocate)
 *   @offsets: offsets (in the blocks) to store the pointers to next.
 *   @branch: place to store the chain in.
 *
 *   This function allocates @num blocks, zeroes out all but the last one,
 *   links them into chain and (if we are synchronous) writes them to disk.
 *   In other words, it prepares a branch that can be spliced onto the
 *   inode. It stores the information about that chain in the branch[], in
 *   the same format as ext2_get_branch() would do. We are calling it after
 *   we had read the existing part of chain and partial points to the last
 *   triple of that (one with zero ->key). Upon the exit we have the same
 *   picture as after the successful ext2_get_block(), excpet that in one
 *   place chain is disconnected - *branch->p is still zero (we did not
 *   set the last link), but branch->key contains the number that should
 *   be placed into *branch->p to fill that gap.
 *
 *   If allocation fails we free all blocks we've allocated (and forget
 *   ther buffer_heads) and return the error value the from failed
 *   ext2_alloc_block() (normally -ENOSPC). Otherwise we set the chain
 *   as described above and return 0.
 */

static int ext2_alloc_branch(struct inode *inode,
                   int num,
                   unsigned long goal,
                   int *offsets,
                   Indirect *branch)
{
     int blocksize = inode->i_sb->s_blocksize;
     int n = 0;
     int err;
     int i;
     int parent = ext2_alloc_block(inode, goal, &err);

     branch[0].key = cpu_to_le32(parent);
     if (parent) for (n = 1; n < num; n++) {
          struct buffer_head *bh;
          /* Allocate the next block */
          int nr = ext2_alloc_block(inode, parent, &err);
          if (!nr)
               break;
          branch[n].key = cpu_to_le32(nr);
          /*
           * Get buffer_head for parent block, zero it out and set
           * the pointer to new one, then send parent to disk.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
398          */
399         bh = getblk(inode->i_dev, parent, blocksize);
400         if (!buffer_uptodate(bh))
401             wait_on_buffer(bh);
402         memset(bh->b_data, 0, blocksize);
403         branch[n].bh = bh;
404         branch[n].p = (u32*) bh->b_data + offsets[n];
405         *branch[n].p = branch[n].key;
406         mark_buffer_uptodate(bh, 1);
407         mark_buffer_dirty_inode(bh, inode);
408         if (IS_SYNC(inode) || inode->u.ext2_i.i_osync) {
409             ll_rw_block (WRITE, 1, &bh);
410             wait_on_buffer (bh);
411         }
412         parent = nr;
413     }
414     if (n == num)
415         return 0;
416 
417     /* Allocation failed, free what we already allocated */
418     for (i = 1; i < n; i++)
419         bforget(branch[i].bh);
420     for (i = 0; i < n; i++)
421         ext2_free_blocks(inode, le32_to_cpu(branch[i].key), 1);
422     return err;
423 }
```

参数 num 表示还有几层映射需要建立，实际上也就是一共需要分配几个记录块，指针 branch 指向前面的数组 chain[]中从映射断裂处开始的那一部分，offset 则指向数组 offsets[]中的相应部分。例如，假若具体的映射是三重间接映射，而在第二层间接映射表中发现相应表项为 0，那么 branch 指向 chain[2]而 offset 指向 offsets[2]，num 则为 2，此时需要分配的是用于第三层间接映射表的记录块以及目标记录块。从某种意义上，分配记录块和建立映射的过程可以看作是对这两个数组的“修复”，是在完成 ext2_get_branch()和 ext2_block_to_path()未竟的事业。注意代码中的 branch[0]表示断裂点的 Indirect 结构，所以是顺着映射的路线“自顶向下”逐层地通过 ext2_alloc_block()在设备上分配记录块和建立映射。

除最底层的记录块，即目标记录块以外，其它的记录块（见代码中的 for 循环）都要通过 get_block()为其在内存中分配缓冲区，并通过 memset()将其缓冲区清成全 0，然后在该缓冲区中建立起本层的映射（403～405 行），再把它标志成“脏”。如果要求同步操作的话，还要立即调用 ll_rw_block()把它写回到设备上。注意代码中的 for 循环里面为之分配缓冲区的是 parent，这都是用于间接映射的记录块，而不是位于最底层的目标记录块。

那么为什么目标记录块是个例外，不需要为其分配缓冲区呢？因为它的缓冲区在调用 ext2_get_block()之前就已经存在了，并且在调用 ext2_get_block()时把指向这个 buffer_head 结构的指针作为参数传了下来；而 ext2_get_block()需要做的就是找到目标记录块的块号，把它设置到这个 buffer_head 结构的 b_blocknr 字段中。前面，对于成功的映射，即 ext2_get_branch()返回 NULL 时，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

ext2_get_block()已经在其标号 got_it 处（见 525 行）这样做了，读者不妨回过去看看。另一方面，在目标记录块的缓冲区中当然不需要再建立什么映射。

还要注意到，在顶层，即原来映射开始断开的那一层上（代码中的 branch[0]），所分配的记录块号只是记入了这一层 Indirect 结构中的 key 字段，却并未写入相应的映射表表项中（由指针 p 所指之处）。就好像我们有了一根树枝，但是还没有使它长在树上。

函数 ext2_alloc_block()的代码也在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 85 123 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>ext2_alloc_block()]

```
static int ext2_alloc_block (struct inode * inode, unsigned long goal, int *err)
{
#ifdef EXT2FS_DEBUG
     static unsigned long alloc_hits = 0, alloc_attempts = 0;
#endif
     unsigned long result;


#ifdef EXT2_PREALLOCATE
     /* Writer: ->i_prealloc* */
     if (inode->u.ext2_i.i_prealloc_count &&
         (goal == inode->u.ext2_i.i_prealloc_block ||
          goal + 1 == inode->u.ext2_i.i_prealloc_block))
     {
          result = inode->u.ext2_i.i_prealloc_block++;
          inode->u.ext2_i.i_prealloc_count--;
          /* Writer: end */
#ifdef EXT2FS_DEBUG
          ext2_debug ("preallocation hit (%lu/%lu).\n",
                  ++alloc_hits, ++alloc_attempts);
#endif
     } else {
          ext2_discard_prealloc (inode);
#ifdef EXT2FS_DEBUG
          ext2_debug ("preallocation miss (%lu/%lu).\n",
                  alloc_hits, ++alloc_attempts);
#endif
          if (S_ISREG(inode->i_mode))
               result = ext2_new_block (inode, goal,
                     &inode->u.ext2_i.i_prealloc_count,
                     &inode->u.ext2_i.i_prealloc_block, err);
          else
               result = ext2_new_block (inode, goal, 0, 0, err);
     }
#else
     result = ext2_new_block (inode, goal, 0, 0, err);
#endif
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    return result;
}
```

参数 goal 表示建议分配的（或要求分配的）设备上记录块号，函数的返回值则为实际分配的块号。内核在编译时有个选择项 EXT2_PREALLOCATE，使文件系统可以“预分配”若干记录块，ext2_inode_info 结构中的 i_prealloc_block 和 i_prealloc_count 两个字段即用于这个目的。我们假定并未采用这个选择项，所以就只剩下对 ext2_new_block()的调用，这个函数的代码在 fs/ext2/balloc.c 中。可是，ext2_new_block()的代码很长，有 250 行以上，而逻辑却并不复杂，所以我们把它留给读者，这里只给出一些简短的说明。

分配时首先试图满足“顾客”的要求，如果所建议的记录块还空闲着就把它分配出去。否则，如果所建议的记录块已经分配掉了，就试图在它附近 32 个记录块的范围内分配。还不行就向前在本块组的位图中搜索，先找位图中整个字节都是 0，即至少有连续 8 个记录块空闲的区间，若达不到目的再降格以求。最后，如果实在找不到，就在整个设备的范围内寻找和分配。

前面说过，除目标记录块以外，对分配的其余记录块都要通过 getblk()为其在内存中分配缓冲区这个函数的代码在 fs/buffer.c 中：

==================== fs/buffer.c 968 1027 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()]

```
/*
 * Ok, this is getblk, and it isn't very clear, again to hinder
 * race-conditions. Most of the code is seldom used, (ie repeating),
 * so it should be much more efficient than it looks.
 *
 * The algorithm is changed: hopefully better, and an elusive bug removed.
 *
 * 14.02.92: changed it to sync dirty buffers a bit: better performance
 * when the filesystem starts to get full of dirty blocks (I hope).
 */
struct buffer_head * getblk(kdev_t dev, int block, int size)
{
    struct buffer_head * bh;
    int isize;

repeat:
    spin_lock(&lru_list_lock);
    write_lock(&hash_table_lock);
    bh = __get_hash_table(dev, block, size);
    if (bh)
        goto out;

    isize = BUFSIZE_INDEX(size);
    spin_lock(&free_list[isize].lock);
    bh = free_list[isize].list;
    if (bh) {
        __remove_from_free_list(bh, isize);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                atomic_set(&bh->b_count, 1);
        }
        spin_unlock(&free_list[isize].lock);

        /*
         * OK, FINALLY we know that this buffer is the only one of
         * its kind, we hold a reference (b_count>0), it is unlocked,
         * and it is clean.
         */
        if (bh) {
                init_buffer(bh, NULL, NULL);
                bh->b_dev = dev;
                bh->b_blocknr = block;
                bh->b_state = 1 << BH_Mapped;

                /* Insert the buffer into the regular lists */
                __insert_into_queues(bh);
        out:
                write_unlock(&hash_table_lock);
                spin_unlock(&lru_list_lock);
                touch_buffer(bh);
                return bh;
        }

        /*
         * If we block while refilling the free list, somebody may
         * create the buffer first ... search the hashes again.
         */
        write_unlock(&hash_table_lock);
        spin_unlock(&lru_list_lock);
        refill_freelist(size);
        goto repeat;
}
```

这里的参数 block 为设备上块号。首先在杂凑表队列中查找，因为这个记录块虽是新分配的，以前为其分配的缓冲区却有可能还在。如不成功则试图从 free_list[]的相应队列中分配。如果分配成功就加以初始化并通过__insert_into_queues()链入相应的杂凑表队列和 LRU 队列（fs/buffer.c）：

==================== fs/buffer.c 494 500 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()__insert_into_queues()]

```
494 static void __insert_into_queues(struct buffer_head *bh)
495 {
496         struct buffer_head **head = &hash(bh->b_dev, bh->b_blocknr);
497 
498         __hash_link(bh, head);
499         __insert_into_lru_list(bh, bh->b_list);
500 }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

当然，从 free_list[]分配缓冲区有可能失败，那就要通过 refill_freelist()再增添一些或者回收一些缓冲区以供周转，其代码在 fs/buffer.c 中：

==================== fs/buffer.c 755 766 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getbtk()>refill_freelist()]

```
/*
 * We used to try various strange things. Let's not.
 * We'll just try to balance dirty buffers, and possibly
 * launder some pages.
 */
static void refill_freelist(int size)
{
	balance_dirty(NODEV);
	if (free_shortage())
		page_launder(GFP_BUFFER, 0);
	grow_buffers(size);
}
```

读者将会看到，对文件的写操作是分两步到位的。第一步是将内容写入缓冲页面中，使缓冲页面成为“脏”页面，然后就把“脏”页面链入一个 LRU 队列，把它“提交”给内核线程 bdflush；第二步是由 bdflush 将已经变“脏”的页面写入文件所在的设备。然后，如果有必要，这些内存页面就可以回收了。内核线程 bdflush 的主体是一个无限循环，平时总在睡眠，每次被唤醒就“冲刷”一次“脏”页面，然后又进入睡眠。但是，为了提高效率，并不是只要有了一个“脏”页面就唤醒 bdflush，而是要到积累起一定数量的“脏”页面时，或者每过一段时间才唤醒它。函数 balance_dirty()的作用就是检查是否已经积累起太多的“脏”页面，如果积累太多了，就把 bdflush 唤醒，其代码在 fs/buffer.c 中：

==================== fs/buffer.c 1064 1078 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()>refill_freelist()>balance_dirty()]

```
/*
 * if a new dirty buffer is created we need to balance bdflush.
 *
 * in the future we might want to make bdflush aware of different
 * pressures on different devices - thus the (currently unused)
 * 'dev' parameter.
 */
void balance_dirty(kdev_t dev)
{
	int state = balance_dirty_state(dev);

	if (state < 0)
		return;
	wakeup_bdflush(state);
}
```

先通过 balance_dirty_stat()检查是否因为已经积累起太多“脏”页面而应该唤醒 bdflush。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/buffer.c 1029 1062 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()>refill_freelist()>balance_dirty()>balance_dirty_state()]

```
/* -1 -> no need to flush
    0 -> async flush
    1 -> sync flush (wait for I/O completation) */
int balance_dirty_state(kdev_t dev)
{
     unsigned long dirty, tot, hard_dirty_limit, soft_dirty_limit;
     int shortage;

     dirty = size_buffers_type[BUF_DIRTY] >> PAGE_SHIFT;
     tot = nr_free_buffer_pages();

     dirty *= 100;
     soft_dirty_limit = tot * bdf_prm.b_un.nfract;
     hard_dirty_limit = tot * bdf_prm.b_un.nfract_sync;

     /* First, check for the "real" dirty limit. */
     if (dirty > soft_dirty_limit) {
          if (dirty > hard_dirty_limit)
               return 1;
          return 0;
     }

     /*
      * If we are about to get low on free pages and
      * cleaning the inactive_dirty pages would help
      * fix this, wake up bdflush.
      */
     shortage = free_shortage();
     if (shortage && nr_inactive_dirty_pages > shortage &&
               nr_inactive_dirty_pages > freepages.high)
          return 0;

     return -1;
}
```

如代码中的注释所述，函数的返回值表明可分配页面的短缺程度。返回-1，表示“脏”页面的数量还不多，因而不需要唤醒 bdflush；返回 0，表示虽然已经积累起相当数量的“脏”页面，但还不是很多，可以让 bdflush 异步地冲刷而不需要停下来等待；返回 1 则表示“脏”页面的数量已经很大，不但要唤醒 bdflush，而且当前进程需要停下来等待其完成，因为此时即使继续往前也多半分配不到空闲页面了。不过，在具体实现的时候又作了一些优化。这是因为：一来不知道 bdflush 与当前进程的优先级孰高孰低，如果 bdflush 的优先级比当前进程的低则即使唤醒了也调度不上；二来既然急着要用空闲页面，需求量又不大，还不如“自己动手，丰衣足食”，先直接冲刷出若干“脏”页面，然后再让 bdflush 继续慢慢冲刷。这样，将这个函数的返回值用作调用 wakeup_bdflush()的参数，就决定了在唤醒 bdflush 以后是否直接调用 flush_dirty_buffers()。

==================== fs/buffer.c 2591 2601 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()>refill_freelist()>balance_dirty()>wakeup_bdflush()]

```
struct task_struct *bdflush_tsk = 0;

void wakeup_bdflush(int block)
{
    if (current != bdflush_tsk) {
        wake_up_process(bdflush_tsk);

        if (block)
            flush_dirty_buffers(0);
    }
}
```

这里的全局量指针 bdflush_tsk 在初始化时设置成指向 bdflush 的 task_struct 结构。这里的 wake_up_process()是个 inline 函数，它将目标进程唤醒，并通过 reschedule_idle()比较目标进程和当前进程的综合权值，如果目标进程的权值更高就把当前进程的 need_schedule 字段设成 1，请求一次调度（详见第 4 章）。然后就根据参数的值决定是否直接调用 flush_dirty_buffers()，其代码在 fs/buffer.c 中：

==================== fs/buffer.c 2530 2589 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()>refill_freelist()>balance_dirty()>wakeup_bdflush()>flush_dirty_buffers()]

```
/* ====================== bdflush support =================== */

/* This is a simple kernel daemon, whose job it is to provide a dynamic
 * response to dirty buffers.  Once this process is activated, we write back
 * a limited number of buffers to the disks and then go back to sleep again.
 */

/* This is the _only_ function that deals with flushing async writes
   to disk.
   NOTENOTENOTENOTE: we _only_ need to browse the DIRTY lru list
   as all dirty buffers lives _only_ in the DIRTY lru list.
   As we never browse the LOCKED and CLEAN lru lists they are infact
   completly useless. */
static int flush_dirty_buffers(int check_flushtime)
{
    struct buffer_head * bh, *next;
    int flushed = 0, i;

 restart:
    spin_lock(&lru_list_lock);
    bh = lru_list[BUF_DIRTY];
    if (!bh)
        goto out_unlock;
    for (i = nr_buffers_type[BUF_DIRTY]; i-- > 0; bh = next) {
        next = bh->b_next_free;
```

```
2555
2556            if (!buffer_dirty(bh)) {
2557                __refile_buffer(bh);
2558                continue;
2559            }
2560            if (buffer_locked(bh))
2561                continue;
2562
2563            if (check_flushtime) {
2564                /* The dirty lru list is chronologically ordered so
2565                   if the current bh is not yet timed out,
2566                   then also all the following bhs
2567                   will be too young. */
2568                if (time_before(jiffies, bh->b_flushtime))
2569                    goto out_unlock;
2570            } else {
2571                if (++flushed > bdf_prm.b_un.ndirty)
2572                    goto out_unlock;
2573            }
2574
2575            /* OK, now we are committed to write it out. */
2576            atomic_inc(&bh->b_count);
2577            spin_unlock(&lru_list_lock);
2578            ll_rw_block(WRITE, 1, &bh);
2579            atomic_dec(&bh->b_count);
2580
2581            if (current->need_resched)
2582                schedule();
2583            goto restart;
2584        }
2585  out_unlock:
2586      spin_unlock(&lru_list_lock);
2587
2588      return flushed;
2589 }
```

为了不至于扯得太远，我们把这段代码留给读者在看完了本章以后回过来自行阅读，注意 2581 行的 current->need_resched 是在前面 wake_up_process()中根据 bdflush 和当前进程的优先级相对大小而设置的。

冲刷一个“脏”页面的结果是把它的内容写回到文件中，为内存页面的回收创造了条件，但是并不等于已经回收了页面。另一方面，只要内存页面不很短缺，则保留这些页面的内容为可能发生的进一步读写提供了缓冲，有利于提高效率。所以，同到 refill_freelist()的代码以后，接着（见前面的 763 行～764 行）就根据系统中页面短缺的程度决定是否调用 page_launder()，详情可参考第 2 章中的有关内容。

最后通过 grow_buffers()再分配若干页面，制造出一些缓冲区来，现在条件已经具备了。我们把 grow_buffers()的代码列在这里让读者自己阅读。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/buffer.c 2244 2305 ====================

[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_alloc_branch()>getblk()>refill_freelist()>grow_buffers()]

```
/*
 * Try to increase the number of buffers available: the size argument
 * is used to determine what kind of buffers we want.
 */
static int grow_buffers(int size)
{
    struct page * page;
    struct buffer_head *bh, *tmp;
    struct buffer_head * insert_point;
    int isize;

    if ((size & 511) || (size > PAGE_SIZE)) {
        printk("VFS: grow_buffers: size = %d\n",size);
        return 0;
    }

    page = alloc_page(GFP_BUFFER);
    if (!page)
        goto out;
    LockPage(page);
    bh = create_buffers(page, size, 0);
    if (!bh)
        goto no_buffer_head;

    isize = BUFSIZE_INDEX(size);

    spin_lock(&free_list[isize].lock);
    insert_point = free_list[isize].list;
    tmp = bh;
    while (1) {
        if (insert_point) {
            tmp->b_next_free = insert_point->b_next_free;
            tmp->b_prev_free = insert_point;
            insert_point->b_next_free->b_prev_free = tmp;
            insert_point->b_next_free = tmp;
        } else {
            tmp->b_prev_free = tmp;
            tmp->b_next_free = tmp;
        }
        insert_point = tmp;
        if (tmp->b_this_page)
            tmp = tmp->b_this_page;
        else
            break;
    }
    tmp->b_this_page = bh;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	free_list[isize].list = bh;
	spin_unlock(&free_list[isize].lock);

	page->buffers = bh;
	page->flags &= ~(1 << PG_referenced);
	lru_cache_add(page);
	UnlockPage(page);
	atomic_inc(&buffermem_pages);
	return 1;

no_buffer_head:
	UnlockPage(page);
	page_cache_release(page);
out:
	return 0;
}
```

结束了 ext2_alloc_branch()的执行，回到 ext2_get_block()中时，我们已经在设备上分配了所需的记录块，包括用于间接映射的中间记录块，但是原先映射开始断开的最高层上所分配的记录块号只是记入了其 Indirect 结构中的 key 字段，却并未写入相应的映射表中。现在就要把“树枝”接到树上（将来，随着文件内容的扩展，这树枝会长成子树）。同时，还需要对所属 inode 结构中的有关内容作一些调整。这些都是由 ext2_splice_branch()完成的，其代码在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 425 491 ====================
[sys_write()>generic_file_write()>ext2_prepare_write()>block_prepare_write()>__block_prepare_write()>ext2_get_block()>ext2_splice_branch()]

```
/**
 *	ext2_splice_branch - splice the allocated branch onto inode.
 *	@inode: owner
 *	@block: (logical) number of block we are adding
 *	@chain: chain of indirect blocks (with a missing link - see
 *		ext2_alloc_branch)
 *	@where: location of missing link
 *	@num:	number of blocks we are adding
 *
 *	This function verifies that chain (up to the missing link) had not
 *	changed, fills the missing link and does all housekeeping needed in
 *	inode (->i_blocks, etc.). In case of success we end up with the full
 *	chain to new block and return 0. Otherwise (== chain had been changed)
 *	we free the new blocks (forgetting their buffer_heads, indeed) and
 *	return -EAGAIN.
 */

static inline int ext2_splice_branch(struct inode *inode,
				long block,
				Indirect chain[4],
				Indirect *where,
				int num)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
447 {
448       int i;
449
450       /* Verify that place we are splicing to is still there and vacant */
451
452       /* Writer: pointers, ->i_next_alloc*, ->i_blocks */
453       if (!verify_chain(chain, where-1) || *where->p)
454            /* Writer: end */
455            goto changed;
456
457       /* That's it */
458
459       *where->p = where->key;
460       inode->u.ext2_i.i_next_alloc_block = block;
461       inode->u.ext2_i.i_next_alloc_goal = le32_to_cpu(where[num-1].key);
462       inode->i_blocks += num * inode->i_sb->s_blocksize/512;
463
464       /* Writer: end */
465
466       /* We are done with atomic stuff, now do the rest of housekeeping */
467
468       inode->i_ctime = CURRENT_TIME;
469
470       /* had we spliced it onto indirect block? */
471       if (where->bh) {
472            mark_buffer_dirty_inode(where->bh, inode);
473            if (IS_SYNC(inode) || inode->u.ext2_i.i_osync) {
474                 ll_rw_block (WRITE, 1, &where->bh);
475                 wait_on_buffer(where->bh);
476            }
477       }
478
479       if (IS_SYNC(inode) || inode->u.ext2_i.i_osync)
480            ext2_sync_inode (inode);
481       else
482            mark_inode_dirty(inode);
483       return 0;
484
485 changed:
486       for (i = 1; i < num; i++)
487            bforget(where[i].bh);
488       for (i = 0; i < num; i++)
489            ext2_free_blocks(inode, le32_to_cpu(where[i].key), 1);
490       return -EAGAIN;
491 }
```

这里的第 459 行将原来映射开始断开的那一层上所分配的记录块号写入了相应的映射表中。这个映射表也许就是 inode 结构中（确切地说是 ext2_inode_info 结构中）的数组 i_data[]，也许是一个用于

PDF-XCHANGE Click to buy NOW! www.docu-track.com

间接映射的记录块。如果相应 Indirect 结构中的指针 bh 为 0（必定是 chain[0]），则映射表就在 inode 结构中。否则，就一定是个间接映射表，因此在改变了其内容以后要将其标志成“脏”。如果要求同步写，则还要立即把它写回设备。

又同到 ext2_get_block()中，现在已经万事俱备了。转到标号 got_it 处，把映射后的记录块号连同设备号置入 bh_result 所指的缓冲区结构中，就完成了任务。有了这些信息，将来就可以把缓冲区的内容写到设备上了。

从 ext2_get_block() 返回，就回到了 __block_prepare_write() 中的第 1586 行。对于 __block_prepare_write()而言，ext2_get_block()为其完成了从文件内块号到设备上块号的映射，这个目标记录块也许是新的，也许原来就存在。如果目标记录块是一个新分配的记录块，就不存在缓冲区的内容与设备上的内容是否一致的问题。但是如果内存中的某一个其它缓冲区仍持有该记录块以前的内容，并且还在杂凑表的某个队列中，则要将那个缓冲区从杂凑队列中脱链并释放。这是通过 unmap_underlying_metadata()完成的。反之，如果目标记录块是原已存在的记录块，则仍有内容是否一致的问题，如果不一致就要先通过 ll_rw_block()从设备上读入。这样，当__block_prepare_write()中的 for 循环结束时，所有涉及本次写操作的物理记录块（缓冲区）都已找到，需要从设备上读入的则已经向设备驱动层发出读入记录块的命令。通过 wait_on_buffer()等待这些命令执行完毕（见 1616 行～1621 行）以后，写操作的准备工作就完成了。

由于__block_prepare_write()是 block_prepare_write()的主体，一旦从前者返回，后者也就结束了，而后者又实际上就是 ext2_prepare_write()，所以就返回到了 generic_file_write()。

在 generic_file_write()中是在一个 while 循环中通过由具体文件系统所提供的函数为写文件操作作准备的。准备好了以后就可以从用户空间把待写的内容复制到缓冲区中，实际上是缓冲页面中。为方便读者阅读，我们再把 while 循环体中的一个片段列出：

==================== mm/filemap.c 2533 2541 ====================

[sys_write()>generic_file_write()]

```
		status = mapping->a_ops->prepare_write(file, page, offset, offset+bytes);
		if (status)
			goto unlock;
		kaddr = page_address(page);
		status = copy_from_user(kaddr+offset, buf, bytes);
		flush_dcache_page(page);
		if (status)
			goto fail_write;
		status = mapping->a_ops->commit_write(file, page, offset, offset+bytes);
```

为写操作作好了准备以后，从缓冲区（缓冲页面）到设备上的记录块这条路就畅通了。这样才可以从用户空间把待写的内容复制过来。

如前所述，目标记录块的缓冲区在文件层是作为缓冲页面的一部分而存在的，所以这是从用户空间到缓冲页面的拷贝，具体通过 copy_from_user()完成。这里 buf 指向用户空间的缓冲区，而(kaddr + offset)为缓冲页面中的起始地址，bytes 则为该页面中待拷贝的长度，这些都是在 while 循环的开头计算好了的。对于 i386 结构的处理器，flush_dcache_page()是空操作。

写入缓冲页面以后，还要把这些缓冲页面提交给内核线程 kflushd，这样写操作才算完成。至于

kflushd 是否来得及马上将这些记录块写回设备上，那是另一回事了。这个将缓冲页面提交给 kflushd 的操作也是因文件系统而异的，由具体文件系统通过其 address_space_operations 结构中的函数指针 commit_write 提供，对于 Ext2 文件系统，这个函数是 generic_commit_write()，其代码在 fs/buffer.c 中：

==================== fs/buffer.c 1844 1856 ====================

[sys_write()>generic_file_write()>generic_commit_write()]

```
int generic_commit_write(struct file *file, struct page *page,
        unsigned from, unsigned to)
{
    struct inode *inode = page->mapping->host;
    loff_t pos = ((loff_t)page->index << PAGE_CACHE_SHIFT) + to;
    __block_commit_write(inode,page,from,to);
    kunmap(page);
    if (pos > inode->i_size) {
        inode->i_size = pos;
        mark_inode_dirty(inode);
    }
    return 0;
}
```

其主体__block_commit_write()的代码也在同一文件中，而 kunmap()对于 i386 结构的处理器为空操作。

==================== fs/buffer.c 1627 1665 ====================

[sys_write()>generic_file_write()>generic_cmomit_write()>__block_commit_write()]

```
static int __block_commit_write(struct inode *inode, struct page *page,
        unsigned from, unsigned to)
{
    unsigned block_start, block_end;
    int partial = 0, need_balance_dirty = 0;
    unsigned blocksize;
    struct buffer_head *bh, *head;

    blocksize = inode->i_sb->s_blocksize;

    for(bh = head = page->buffers, block_start = 0;
       bh != head || !block_start;
       block_start=block_end, bh = bh->b_this_page) {
        block_end = block_start + blocksize;
        if (block_end <= from || block_start >= to) {
            if (!buffer_uptodate(bh))
                partial = 1;
        } else {
            set_bit(BH_Uptodate, &bh->b_state);
            if (!atomic_set_buffer_dirty(bh)) {
                __mark_dirty(bh);
                buffer_insert_inode_queue(bh, inode);
                need_balance_dirty = 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                }
            }
        }

        if (need_balance_dirty)
            balance_dirty(bh->b_dev);
        /*
         * is this a partial write that happened to make all buffers
         * uptodate then we can optimize away a bogus readpage() for
         * the next read(). Here we 'discover' wether the page went
         * uptodate as a result of this (potentially partial) write.
         */
        if (!partial)
            SetPageUptodate(page);
        return 0;
}
```

函数中的 for 循环扫描缓冲页面中的每个记录块，如果一个记录块与写入的范围（从 from 到 to）相交，就把该记录块的缓冲区设成“ uptodate”，即与设备上的记录块相一致，并将其标志成 dirty，下面的事就交给 kflushd 了。值得注意的是这里已经将缓冲区的 BH_Uptodate 标志位设成 1，表示缓冲区的内容已经与设备上相一致。可是，实际上此时缓冲区的内容尚未写回设备，所以从物理上说显然是不一致的。但是，由于写操作本身已接近完成，涉及的缓冲区即将提交给 kflushd，从逻辑的角度上缓冲区中的内容与设备上的内容已经一致了。所以所谓“一致”或“不一致”只是一个逻辑上的概念，而并非物理上的概念。只要写入的内容已经“提交”（commit），就认为已经一致了。而不一致的状态只发生在写操作的中途，即改变了缓冲区（或部分缓部区）的内容而尚未提交之前。在写入的准备阶段，遇有不一致的缓冲区就要从设备上重新读入，就是因为有未完成的写操作存在而破坏了缓冲区的内容。此外，在将缓冲区设置成 dirty 时，如果该缓冲区原来是“干净”的，那么一来要调用__mark_dirty()，二来要将 need_balance_dirty 设成 1。调用__mark_dirty()的目的是将缓冲区根据具体情况转移到合适的 LRU 队列中，有关的代码均在文件 fs/buffer.c 中：

==================== fs/buffer.c 1080 1084 ====================

[sys_write()>generic_file_write()>generic_cmomit_write()>__block_commit_write()>__mark_dirty()]

```
static __inline__ void __mark_dirty(struct buffer_head *bh)
{
        bh->b_flushtime = jiffies + bdf_prm.b_un.age_buffer;
        refile_buffer(bh);
}
```

==================== fs/buffer.c 1124 1129 ====================

[sys_write()>generic_file_write()>generic_cmomit_write()>__block_commit_write()>__mark_dirty()>refile_buffer()]

```
void refile_buffer(struct buffer_head *bh)
{
        spin_lock(&lru_list_lock);
        __refile_buffer(bh);
        spin_unlock(&lru_list_lock);
}
```

==================== fs/buffer.c 1102 1122 ====================

[sys_write()>generic_file_write()>generic_cmomit_write()>__block_commit_write()>__mark_dirty()>refile_buffer()]

```
/*
 * A buffer may need to be moved from one buffer list to another
 * (e.g. in case it is not shared any more). Handle this.
 */
static void __refile_buffer(struct buffer_head *bh)
{
	int dispose = BUF_CLEAN;
	if (buffer_locked(bh))
		dispose = BUF_LOCKED;
	if (buffer_dirty(bh))
		dispose = BUF_DIRTY;
	if (buffer_protected(bh))
		dispose = BUF_PROTECTED;
	if (dispose != bh->b_list) {
		__remove_from_lru_list(bh, bh->b_list);
		bh->b_list = dispose;
		if (dispose == BUF_CLEAN)
			remove_inode_queue(bh);
		__insert_into_lru_list(bh, dispose);
	}
}
```

数据结构 buffer_head 通过其指针 b_next_free 和 b_prev_free 链入到空闲缓冲区队列或某个 LRU 队列中，而作为记录块缓冲区 LRU 队列头部的 lru_list[]则是一个指针数组，其定义见文件 fs/buffer.c：

==================== fs/buffer.c 82 82 ====================

```
static struct buffer_head *lru_list[NR_LIST];
```

这个数组是以记录块缓冲区的状态为下标的（include/linux/fs.h）。

==================== include/linux/fs.h 991 995 ====================

```
#define BUF_CLEAN  0
#define BUF_LOCKED 1    /* Buffers scheduled for write */
#define BUF_DIRTY  2    /* Dirty buffers, not yet scheduled for write */
#define BUF_PROTECTED   3    /* Ramdisk persistent storage */
#define NR_LIST         4
```

这样，对处于各种不同状态的记录块缓冲区，就各自有一个 LRU 队列，而 bdflush 就只扫描 lru_list[BUF_DIRTY]队列。

最后，只要有记录块缓冲区从“干净”状态变成“脏”状态，也就是如果 need_balance_dirty 为 1，就要通过 balance_dirty()看看这样的记录块是否已经积累到了一定的数量，如果是，就唤醒 bdflush 进行一次“冲刷”。这个函数的代码已经在前面看到过了。

不管是否立即唤醒 bdflush，总之此后的事情就交给它了。我们将在设备驱动一章中回到这个话题上来。

完成了 generic_commit_write()以后，generic_file_write()中的一轮循环，也就是对一个缓冲页面的写入就完成了。从而对该页面的使用也结束了，所以要通过 page_cache_release()递减对该页面的使用

PDF-XCHANGE Click to buy NOW! www.docu-track.com

计数。

总结对一个缓冲页面的写文件操作，大致可以分为三个阶段。第一是准备阶段，第二是缓冲页面的写入阶段，最后是提交阶段。完成了对所涉及的所有页面的循环，整个写文件操作的主体 generic_file_write()就告结束，并且 sys_write()也随着结束了。

理解了 sys_write()，再来看 sys_read()就容易一些了。这两个函数几乎是一样的，只是在 sys_write()中要验证用户空间的缓冲区可读，并且使用 file_operations 结构中的函数指针 write，而在 sys_read()中则要验证用户空间的缓冲区可写，并且使用 file_operations 结构中的函数指针 read。就 Ext2 文件系统的读操作而言，这个函数指针指向 generic_file_read()，其代码也在 mm/filemap.c 中：

==================== mm/filemap.c 1237 1264 ====================

[sys_read()>generic_file_read()]

```
/*
 * This is the "read()" routine for all filesystems
 * that can use the page cache directly.
 */
ssize_t generic_file_read(struct file * filp, char * buf, size_t count, loff_t *ppos)
{
	ssize_t retval;

	retval = -EFAULT;
	if (access_ok(VERIFY_WRITE, buf, count)) {
		retval = 0;

		if (count) {
			read_descriptor_t desc;

			desc.written = 0;
			desc.count = count;
			desc.buf = buf;
			desc.error = 0;
			do_generic_file_read(filp, ppos, &desc, file_read_actor);

			retval = desc.written;
			if (!retval)
				retval = desc.error;
		}
	}
	return retval;
}
```

显然，这个函数只是 do_generic_file_read()的“包装”。其目的在于检查对用户空间缓冲区的写访问权，并为读文件操作准备下一个“读操作描述结构”，即 read_descriptor_t 数据结构，以减少在调用 do_generic_file_read()时传递参数的个数。

由于 do_generic_file_read()的代码比较长，我们还是分段阅读，其代码在同一文件 filemap.c 中：

==================== mm/filemap.c 1005 1065 ====================

[sys_read()>generic_file_read()>do_generic_file_read()]

```
/*
 * This is a generic file read routine, and uses the
 * inode->i_op->readpage() function for the actual low-level
 * stuff.
 *
 * This is really ugly. But the goto's actually try to clarify some
 * of the logic when it comes to error handling etc.
 */
void do_generic_file_read(struct file * filp, loff_t *ppos, read_descriptor_t * desc, read_actor_t actor)
{
     struct inode *inode = filp->f_dentry->d_inode;
     struct address_space *mapping = inode->i_mapping;
     unsigned long index, offset;
     struct page *cached_page;
     int reada_ok;
     int error;
     int max_readahead = get_max_readahead(inode);

     cached_page = NULL;
     index = *ppos >> PAGE_CACHE_SHIFT;
     offset = *ppos & ~PAGE_CACHE_MASK;

/*
 * If the current position is outside the previous read-ahead window,
 * we reset the current read-ahead context and set read ahead max to zero
 * (will be set to just needed value later),
 * otherwise, we assume that the file accesses are sequential enough to
 * continue read-ahead.
 */
     if (index > filp->f_raend || index + filp->f_rawin < filp->f_raend) {
          reada_ok = 0;
          filp->f_raend = 0;
          filp->f_ralen = 0;
          filp->f_ramax = 0;
          filp->f_rawin = 0;
     } else {
          reada_ok = 1;
     }
/*
 * Adjust the current value of read-ahead max.
 * If the read operation stay in the first half page, force no readahead.
 * Otherwise try to increase read ahead max just enough to do the read request.
 * Then, at least MIN_READAHEAD if read ahead is ok,
 * and at most MAX_READAHEAD in all cases.
 */
     if (!index && offset + desc->count <= (PAGE_CACHE_SIZE >> 1)) {
          filp->f_ramax = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    } else {
        unsigned long needed;

        needed = ((offset + desc->count) >> PAGE_CACHE_SHIFT) + 1;

        if (filp->f_ramax < needed)
            filp->f_ramax = needed;

        if (reada_ok && filp->f_ramax < MIN_READAHEAD)
                filp->f_ramax = MIN_READAHEAD;
        if (filp->f_ramax > max_readahead)
            filp->f_ramax = max_readahead;
    }

```

参数 actor 是一个函数指针，这里的实际参数是 file_read_actor()，这个函数的作用是将文件的内容从缓冲页面拷贝到用户空间的缓冲区中。

文件的读操作有一个比写操作更复杂之处，那就是预读。我们在本节开头时曾谈到过预读，现在就要涉及具体的代码了。预读量的大小是与具体设备有关的，内核中设置了一个以主设备号为下标的数组 max_readahead[]，其定义在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 111 114 ====================

```
/*
 * The following tunes the read-ahead algorithm in mm/filemap.c
 */
int * max_readahead[MAX_BLKDEV];
```

数组中的每个元素都是指针，指向以次设备号为下标的另一个整数数组，那个数组中的元素就是每个具体设备的最大预读量。同时，内核中还提供了一个 inline 函数 get_max_readahead()，利用这个函数根据 inode 结构中的设备号就可确定对特定文件的最大预读量。这个函数的定义在 mm/filemap.c 中，代码的作者还为之写了一大段注释，我们把它一并列在下面。

==================== mm/filemap.c 829 890 ====================

[sys_write()>generic_file_read()>do_generic_file_read()>get_max_readahead()]

```
/*
 * Read-ahead context:
 * -------------------
 * The read ahead context fields of the "struct file" are the following:
 * - f_raend : position of the first byte after the last page we tried to
 *           read ahead.
 * - f_ramax : current read-ahead maximum size.
 * - f_ralen : length of the current IO read block we tried to read-ahead.
 * - f_rawin : length of the current read-ahead window.
 *         if last read-ahead was synchronous then
 *              f_rawin = f_ralen
 *         otherwise (was asynchronous)
 *              f_rawin = previous value of f_ralen + f_ralen
 *
```

```
 * Read-ahead limits:
 * ------------------
 * MIN_READAHEAD   : minimum read-ahead size when read-ahead.
 * MAX_READAHEAD   : maximum read-ahead size when read-ahead.
 *
 * Synchronous read-ahead benefits:
 * --------------------------------
 * Using reasonable IO xfer length from peripheral devices increase system
 * performances.
 * Reasonable means, in this context, not too large but not too small.
 * The actual maximum value is:
 *   MAX_READAHEAD + PAGE_CACHE_SIZE = 76k is CONFIG_READA_SMALL is undefined
 *     and 32K if defined (4K page size assumed).
 *
 * Asynchronous read-ahead benefits:
 * ---------------------------------
 * Overlapping next read request and user process execution increase system
 * performance.
 *
 * Read-ahead risks:
 * -----------------
 * We have to guess which further data are needed by the user process.
 * If these data are often not really needed, it's bad for system
 * performances.
 * However, we know that files are often accessed sequentially by
 * application programs and it seems that it is possible to have some good
 * strategy in that guessing.
 * We only try to read-ahead files that seems to be read sequentially.
 *
 * Asynchronous read-ahead risks:
 * ------------------------------
 * In order to maximize overlapping, we must start some asynchronous read
 * request from the device, as soon as possible.
 * We must be very careful about:
 * - The number of effective pending IO read requests.
 *   ONE seems to be the only reasonable value.
 * - The total memory pool usage for the file access stream.
 *   This maximum memory usage is implicitly 2 IO read chunks:
 *   2*(MAX_READAHEAD + PAGE_CACHE_SIZE) = 156K if CONFIG_READA_SMALL is undefined,
 *   64k if defined (4K page size assumed).
 */

static inline int get_max_readahead(struct inode * inode)
{
	if (!inode->i_dev || !max_readahead[MAJOR(inode->i_dev)])
		return MAX_READAHEAD;
	return max_readahead[MAJOR(inode->i_dev)][MINOR(inode->i_dev)];
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的常数 MAX_READAHEAD 定义为 31，即 31 个页面，124K 字节。

如前所述，由于预读的引入，现在 file 结构中要维持两个上下文了。一个是以“当前位置”f_pos 为代表的真正的读／写上下文，另一个则是预读的上下文。为此目的在 file 结构中增设了 f_reada、f_ramax、f_raend、f_ralen 以及 f_rawin 等五个字段。这五个字段的名称反映了它们的用途，代码作者在注释中也作了说明。所谓“预读的上下文”，实际上是一个窗口。窗口的末端就是 f_raend，而窗口的大小则为 f_rawin。与写操作相似，局部量 index 为当前读写位置所在页面的序号，offset 则为页面内的位移。如果读操作的起始页面落在预读窗口的外面，也就是 index 大于预读窗口的终点页面或者小于预读窗口的起始页面，那么现存的预读窗口与当前的读操作就没有什么关系了，所以要另起炉灶来一个新的预读窗口（见 1034～1039 行）。否则就是如何推进现有预读窗口的问题，所以先保持现有的窗口不变，而将局部量 read_ok 设成 1。然后，还要对 file 结构中的最大预读量作一些调整。如果当前所要求的读操作仅仅局限在文件的第一个页面的前半部分中进行（见 1050 行），那就根本不需要预读，所以将 file 结构中的 f_ramax 字段设成 0。否则就要依据整个读操作所涉及的页面数量 needed 和一些常量、参数适当调整 f_ramax 字段的数值（见 1057～1063 行）。对预读操作上下文作了这些准备以后，就开始读了。继续看 do_generic_file_read()的代码：

==================== mm/filemap.c 1066 1125 ====================

[sys_read()>generic_file_read()>do_generic_file_read()]

```
1066      for (;;) {
1067           struct page *page, **hash;
1068           unsigned long end_index, nr;
1069
1070           end_index = inode->i_size >> PAGE_CACHE_SHIFT;
1071           if (index > end_index)
1072                break;
1073           nr = PAGE_CACHE_SIZE;
1074           if (index == end_index) {
1075                nr = inode->i_size & ~PAGE_CACHE_MASK;
1076                if (nr <= offset)
1077                     break;
1078           }
1079
1080           nr = nr - offset;
1081
1082           /*
1083            * Try to find the data in the page cache..
1084            */
1085           hash = page_hash(mapping, index);
1086
1087           spin_lock(&pagecache_lock);
1088           page = __find_page_nolock(mapping, index, *hash);
1089           if (!page)
1090                goto no_cached_page;
1091 found_page:
1092           page_cache_get(page);
1093           spin_unlock(&pagecache_lock);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1094
1095          if (!Page_Uptodate(page))
1096               goto page_not_up_to_date;
1097          generic_file_readahead(reada_ok, filp, inode, page);
1098 page_ok:
1099          /* If users can be writing to this page using arbitrary
1100           * virtual addresses, take care about potential aliasing
1101           * before reading the page on the kernel side.
1102           */
1103          if (mapping->i_mmap_shared != NULL)
1104               flush_dcache_page(page);
1105
1106          /*
1107           * Ok, we have the page, and it's up-to-date, so
1108           * now we can copy it to user space...
1109           *
1110           * The actor routine returns how many bytes were actually used..
1111           * NOTE! This may not be the same as how much of a user buffer
1112           * we filled up (we may be padding etc), so we can only update
1113           * "pos" here (the actor routine has to update the user buffer
1114           * pointers and the remaining count).
1115           */
1116          nr = actor(desc, page, offset, nr);
1117          offset += nr;
1118          index += offset >> PAGE_CACHE_SHIFT;
1119          offset &= ~PAGE_CACHE_MASK;
1120
1121          page_cache_release(page);
1122          if (nr && desc->count)
1123               continue;
1124          break;
1125
```

不难想像，整个读操作是通过一个循环完成的，这个循环依次走过所涉及的每个缓冲页面，完成从这些页面的读出。由于这个 for 循环内部的流程比较复杂，我们通过一个假想的情景来遍历这个 for 循环的代码，这个情景涉及对三个缓冲页面的读出。

与写操作不同，当读操作的位置到达了（或超出了）文件的末尾就结束了（见 1070～1078 行），而不像写操作或 lseek()那样将文件的末尾向前推进。只要还没有到达文件的末尾，就根据页面的大小或者目标文件在其最后一个页面中的大小 nr，以及读操作在当前页面中的起点 offset 计算出从当前页面读出的长度（见 1073～1080 行）。

决定了从当前页面中读出的长度以后，就要设法找到或读入相应的缓冲页面了。首先当然是根据目标页面的杂凑值从杂凑表队列中寻找（见 1085～1088 行）。寻找的结果有三种可能，第一种是找不到，第二种是找到了，但是该缓冲页面的内容不一致，第三种是既找到了所需的缓冲页面，页面的内容又一致。

在我们的情景里，假定第一个缓冲页面找到了，并且一致，所以就到达了第 1098 行的 page_ok 标

号处。既然找到了目标页面，下面的事情就顺理成章了。如前所述，参数 actor 是个函数指针，这个指针实际上指向 file_read_actor()。它的作用就是从缓冲页面把内容复制到用户空间的缓冲区中，并相应调整读操作描述结构中的待读出长度，最后返回已复制的长度。完成了从缓冲页面中的读出以后，就根据 file_read_actor()的返回值 nr 将 index 和 offset 两个变量的值向前推进，并将当前页面释放（递减其使用计数）。在我们这个情景中，从这个页面读出的长度 nr 非 0，尚待读出的长度也还未达到 0，所以经由第 1123 行的 continue 语句开始下一轮循环（否则就经由第 1124 行的 break 语句结束循环）。

我们假定寻找第二个目标页面的结果也找到了，但是页面的内容不一致，所以在第 1096 行转移到标号 page_not_up_to_date 处：

==================== mm/filemap.c 1126 1171 ====================

[sys_read()>generic_file_read()>do_generic_file_read()]

```
/*
 * Ok, the page was not immediately readable, so let's try to read ahead while we're at it..
 */
page_not_up_to_date:
		generic_file_readahead(reada_ok, filp, inode, page);

		if (Page_Uptodate(page))
			goto page_ok;

		/* Get exclusive access to the page ... */
		lock_page(page);

		/* Did it get unhashed before we got the lock? */
		if (!page->mapping) {
			UnlockPage(page);
			page_cache_release(page);
			continue;
		}

		/* Did somebody else fill it already? */
		if (Page_Uptodate(page)) {
			UnlockPage(page);
			goto page_ok;
		}

readpage:
		/* ... and start the actual read. The read will unlock the page. */
		error = mapping->a_ops->readpage(filp, page);

		if (!error) {
			if (Page_Uptodate(page))
				goto page_ok;

			/* Again, try some read-ahead while waiting for the page to finish.. */
			generic_file_readahead(reada_ok, filp, inode, page);
			wait_on_page(page);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1162                if (Page_Uptodate(page))
1163                        goto page_ok;
1164                error = -EIO;
1165        }
1166
1167        /* UHHUH! A synchronous read error occurred. Report it */
1168        desc->error = error;
1169        page_cache_release(page);
1170        break;
1171
```

由于页面的内容不一致，所以不能马上从这个页面读出。页面内容不一致是个暂时的现象，这是由于某个进程正在写包括这个页面在内的某些页面，但尚未提交所造成的。一般只要等待一会儿就行了。可既然要等待，就不如乘机预读一些页面进来，所以通过 generic_file_readahead()启动预读。我们把这个函数的阅读暂时放一下，在这里只要知道这个函数启动预读就行了。不过要注意，这里说的是启动预读，而不是完成预读，实际的页面读入是异步的。

启动了预读以后，再来检查当前的目标页面是否已经一致（见第 1132 行）。如果已经一致了那就转到 page_ok 标号处（第 1098 行），下面就与第一个页面的情况相同了。如果还没有一致呢？那就要从设备上把这个页面读同来。读之前要先把页面锁住，注意这里的 lock_page()可能隐含着等待，因为这页面可能已经被别的进程锁住了。特别是这个页面还不一致，就说明有某个进程正在对其进行写操作，很可能就是这个进程锁住了页面。所以，lock_page()的过程实际上就是睡眠等待当前锁住这个页面的进程完成其操作并且解锁的过程。当从 lock_page()返回时，这个页面已经被当前进程锁住了。正因为这样，就很有可能当加锁成功时页面已经一致了，所以要再次加以检查，如果确已一致，就把锁解除并转向 page_ok。

要是加了锁而页面仍旧没有达到一致，那就无计可施，只好从设备上把页面读回来，这就到了标号 readpage 处。对具体文件系统和设备的读操作是由具体的 address_space_operations 数据结构通过函数指针 readpage 提供的，对于 Ext2 文件系统这个函数是 ext2_readpage()，其代码在 fs/ext2/inode.c 中：

==================== fs/ext2/inode.c 657 660 ====================

[sys_read()>generic_file_read()>do_generic_file_read()>ext2_readpage()]

```
static int ext2_readpage(struct file *file, struct page *page)
{
        return block_read_full_page(page,ext2_get_block);
}
```

这个函数通过一个通用的函数，即 block_read_full_page()完成操作，而以 ext2_get_block()作为调用的参数之一。读者应该还记得，ext2_get_block()完成 Ext2 文件系统从文件中逻辑块号到设备上块号的映射。函数 block_read_full_page()的代码在 fs/buffer.c 中：

==================== fs/buffer.c 1667 1744 ====================

[sys_read()>generic_file_read()>do_generic_file_read()>ext2_readpage()>block_read_full_page()]

```
/*
 * Generic "read page" function for block devices that have the normal
 * get_block functionality. This is most of the block device filesystems.
 * Reads the page asynchronously --- the unlock_buffer() and
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * mark_buffer_uptodate() functions propagate buffer state into the
 * page struct once IO has completed.
 */
int block_read_full_page(struct page *page, get_block_t *get_block)
{
    struct inode *inode = page->mapping->host;
    unsigned long iblock, lblock;
    struct buffer_head *bh, *head, *arr[MAX_BUF_PER_PAGE];
    unsigned int blocksize, blocks;
    int nr, i;

    if (!PageLocked(page))
        PAGE_BUG(page);
    blocksize = inode->i_sb->s_blocksize;
    if (!page->buffers)
        create_empty_buffers(page, inode->i_dev, blocksize);
    head = page->buffers;

    blocks = PAGE_CACHE_SIZE >> inode->i_sb->s_blocksize_bits;
    iblock = page->index << (PAGE_CACHE_SHIFT - inode->i_sb->s_blocksize_bits);
    lblock = (inode->i_size+blocksize-1) >> inode->i_sb->s_blocksize_bits;
    bh = head;
    nr = 0;
    i = 0;

    do {
        if (buffer_uptodate(bh))
            continue;

        if (!buffer_mapped(bh)) {
            if (iblock < lblock) {
                if (get_block(inode, iblock, bh, 0))
                    continue;
            }
            if (!buffer_mapped(bh)) {
                memset(kmap(page) + i*blocksize, 0, blocksize);
                flush_dcache_page(page);
                kunmap(page);
                set_bit(BH_Uptodate, &bh->b_state);
                continue;
            }
            /* get_block() might have updated the buffer synchronously */
            if (buffer_uptodate(bh))
                continue;
        }

        arr[nr] = bh;
        nr++;
    } while (i++, iblock++, (bh = bh->b_this_page) != head);
```

```
1720
1721     if (!nr) {
1722          /*
1723           * all buffers are uptodate - we can set the page
1724           * uptodate as well.
1725           */
1726          SetPageUptodate(page);
1727          UnlockPage(page);
1728          return 0;
1729     }
1730
1731     /* Stage two: lock the buffers */
1732     for (i = 0; i < nr; i++) {
1733          struct buffer_head * bh = arr[i];
1734          lock_buffer(bh);
1735          bh->b_end_io = end_buffer_io_async;
1736          atomic_inc(&bh->b_count);
1737     }
1738
1739     /* Stage 3: start the IO */
1740     for (i = 0; i < nr; i++)
1741          submit_bh(READ, arr[i]);
1742
1743     return 0;
1744 }
```

每个缓冲页面都包含着若干记录块缓冲区，page 数据结构中的 buffer_head 指针 buffers 指向这些缓冲区的 buffer_head 数据结构队列。如果一个缓冲页面尚未建立起这样的队列，就要通过 create_empty_buffers()加以创建。很自然地，然后是对构成该页面的各个记录块缓冲区的循环。以前讲过，一个页面的内容不一致并不说明构成这个页面的所有记录块都不一致。所以，如果一个记录块的内容是一致的就把它跳过（见第 1698 行的 continue 语句）。如果一个记录块缓冲区尚未与设备上的物理记录块建立起映射关系（见第 1700 行），并且这个记录块的起始地址并未超出文件的末尾（见第 1701 行和第 1702 行），就要通过作为参数传递下来的函数建立起映射。在这里，对于 Ext2 文件系统而言，这个函数就是 ext2_get_block()，我们已经在前面读过它的代码。

不过，这里对这个函数的调用与写文件时有所不同，那就是第三个参数为 0，而在写操作时这个参数为 1。这个参数表示如果尚未为给定的逻辑记录块分配物理记录块的话，是否要为之分配一个。我们在前面读 ext2_get_block()的代码时跳过了当这个参数为 0 时的那一部分，现在回过头去看一下。该函数代码中标号 cleanup 前有个 if (!creat...)语句；当 ext2_get_branch()返回了一个非 0 指针，表示尚未为给定的逻辑记录块分配物理记录块时，就由这个 if 语句决定怎么办。如果参数 create 为 0，就表示不为之分配物理记录块，此时并不设置相应缓冲区头部中的有关字段，也并不将其 BH_MAPPED 标志位设置成 1。

所以，如果在调用了 ext2_get_block()以后缓冲区的映射仍未建立，就表示这个逻辑记录块尚无与之对应的物理记录块。这种情况发生在通过 lseek()系统调用在文件中引入了空洞以后。

从空洞中读出的内容是什么呢？请看下面紧接着的几行。就是说，如果逻辑记录块落在一个空洞

PDF-XCHANGE Click to buy NOW! www.docu-track.com

内，就把它清成全 0，所以读出的内容也是全 0。那么，什么时候才为这个逻辑记录块分配设备上的物理空间呢？要等到对这个记录块进行写操作的时候。到那时，调用 ext2_get_block()的第三个参数 create 是 1，就会为之分配物理记录块了。

除这两种情况以外，那就是已经建立起映射但是内容不一致的页面了。这种页面是真正需要从设备上读入的页面，所以一方面通过 init_buffer()对 buffer_head 结构进行一些设置，主要是对函数指针 b_end_io 的设置，这个函数指针提供了当设备的 I/O 完成时要启动的操作，在这里是 end_buffer_io_async()。函数 init_buffer()的代码也在 buffer.c 中：

==================== fs/buffer.c 768 773 ====================

[sys_read()>generic_file_read()>do_generic_file_read()>ext2_readpage()>block_read_full_page()>init_buffer()]

```
void init_buffer(struct buffer_head *bh, bh_end_io_t *handler, void *private)
{
	bh->b_list = BUF_CLEAN;
	bh->b_end_io = handler;
	bh->b_private = private;
}
```

虽然这个记录块是肯定要从设备上读入的，但是却并不立即就在循环体内启动对设备的操作，而只是先把真正需要读入的记录块缓冲区收集在一个指针数组 arr[]中（见第 1733 行）。这个数组随后被作为参数传递给 ll_rw_block()，将积累起来的属于同一页面的记录块成批地读入，而对 ll_rw_block()的调用则留待对记录块的循环结束以后。

记录块的读入是需要一定时间的，而 ll_rw_block()实际上只是启动记录块的读入，所以从 ll_bw_block()以及随之从 ext2_readpage()的返回和读入的完成（通过 DMA 完成）是异步的，互相平行的。

这样，当返回到 do_generic_file_read()中时（第 1155 行），页面中需要读入的记录块也许已经全部完成，从而使页面的 PG_UPTODATE 标志位已经变成了 1，表明该页面的内容已经一致，但是也有可能尚未全部完成而页面的内容尚未一致。如果是前者就转入 page_ok，此后的操作就与前述第一个页面的情况相同了。

可是如果尚未完成呢？那就需要等待。既然要等待，那何不干脆再多读一些记录块进来备用呢？所以这里又调用处理预读的 generic_file_readahead()。对于因页面内容不一致而从标号 page_not_up_to_date 执行下来进入 readpage 的路线而言，这已经是第二次调用 generic_file_readahead()了。但是进入 readpage 的路线并非只有这么一条，所以这里的预读一方面也是出于对其它情况的考虑，这一点读者以后就会看到。

虽然通过预读消耗了一些时间，目标页面的读入仍不能肯定已经完成，所以要通过 wait_on_page()加以检验或等待。到页面（实际上是其中的若干记录块）的读入肯定已经完成时，页面的 PG_uptodate 标志位应该为 1，否则就有错了。这里 Page_Uptodate()所作的仅仅是一种检验而不包含等待，而 wait_on_page()则包含了可能的睡眠等待。

目标页面的读入顺利完成以后，就转向 page_ok，此后的操作就又与前述第一个页面相同了。完成了第二个缓冲页面的读出以后，由于所要求的读出尚未完成，又通过第 1123 行的 continue 语句回到了 for 循环的开头。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这一次，由于涉及的第三个逻辑页面没有被缓冲在内存中，__find_page_nolock()返回 NULL，所以就转到了 no_cached_page。我们在文件 filemap.c 中继续往下看：

==================== mm/filemap.c 1172 1213 ====================

[sys_read()>generic_file_read()>do_generic_file_read()]

```
no_cached_page:
		/*
		 * Ok, it wasn't cached, so we need to create a new
		 * page..
		 *
		 * We get here with the page cache lock held.
		 */
		if (!cached_page) {
			spin_unlock(&pagecache_lock);
			cached_page = page_cache_alloc();
			if (!cached_page) {
				desc->error = -ENOMEM;
				break;
			}

			/*
			 * Somebody may have added the page while we
			 * dropped the page cache lock. Check for that.
			 */
			spin_lock(&pagecache_lock);
			page = __find_page_nolock(mapping, index, *hash);
			if (page)
				goto found_page;
		}

		/*
		 * Ok, add the new page to the hash-queues...
		 */
		page = cached_page;
		__add_to_page_cache(page, mapping, index, hash);
		spin_unlock(&pagecache_lock);
		cached_page = NULL;

		goto readpage;
	}

	*ppos = ((loff_t) index << PAGE_CACHE_SHIFT) + offset;
	filp->f_reada = 1;
	if (cached_page)
		page_cache_free(cached_page);
	UPDATE_ATIME(inode);
}
```

既然在内存中尚未为目标页面建立起缓冲，那就不仅仅是从设备读入的问题了，在此之前还要为

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

之分配一个页面。在前面第 1023 行中指针 cached_page 初始化成 NULL，表示没有已经分配但尚未使用的缓冲页面，所以这里通过 page_cache_alloc()分配一个页面备用。但是，在分配成功以后还要再检查一次目标页面是否已经缓冲（见第 1182 行），这是因为在 page_cache_alloc()中当前进程有可能进入睡眠，从而有可能让别的进程抢先为目标页面建立了缓冲。如果这种情况果真发生了，那就转入 found_page，此后的操作就与前述的第一和第二个页面相同了。至于分配得的页面 cached_page，则成了“已经分配但尚未使用的缓冲页面”，我们不必忙着将其释放，因为也许以后还会有需要，如果确实没有需要就拖延到最后在第 1211 行中加以释放。否则，要是没有发生这样的情况，那就将分配得的页面链入到所有有关的队列中，包括由所属 inode 结构中的指针 i_mapping 所指向的 address_space 结构(通常是 inode 结构中的 i_data）里面的缓冲页面队列、全局性的缓冲页面杂凑表队列以及全局性的缓冲页面 LRU 队列。然后就转向 readpage，此后的操作就与前述第二个页面的一部分操作相同了。

由于这已经是涉及的最后一个页面，所以从这个页面的读出完成以后，就通过第 1124 行的 break 语句结束 for 循环而到达第 1208 行，在这里对 file 结构中的 f_pos 字段加以调整。注意，index 和 offset 的值在循环中每次都在向前推进(见 1117～1119 行)，所以此时已经指向本次 read()操作以后的位置上。另一方面，当前的预读上下文继续有效，所以将 file 结构中的 f_reada 标志设成 1。

由于 do_generic_file_readahead()是 generic_file_read()的主体，至此我们可以认为读操作已经完成。

在上面的叙述中，我们跳过了预读函数 generic_flle_readahead()的细节，这个函数的代码也在 mm/filemap.c 中：

==================== mm/filemap.c 892 1002 ====================

[sys_read()>generic_file_read()>do_gerieric_file_read()>generic_file_readahead()]

```
892 static void generic_file_readahead(int reada_ok,
893      struct file * filp, struct inode * inode,
894      struct page * page)
895 {
896      unsigned long end_index = inode->i_size >> PAGE_CACHE_SHIFT;
897      unsigned long index = page->index;
898      unsigned long max_ahead, ahead;
899      unsigned long raend;
900      int max_readahead = get_max_readahead(inode);
901
902      raend = filp->f_raend;
903      max_ahead = 0;
904
905 /*
906  * The current page is locked.
907  * If the current position is inside the previous read IO request, do not
908  * try to reread previously read ahead pages.
909  * Otherwise decide or not to read ahead some pages synchronously.
910  * If we are not going to read ahead, set the read ahead context for this
911  * page only.
912  */
913      if (PageLocked(page)) {
914           if (!filp->f_ralen || index >= raend || index + filp->f_rawin < raend) {
915                raend = index;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            if (raend < end_index)
                max_ahead = filp->f_ramax;
            filp->f_rawin = 0;
            filp->f_ralen = 1;
            if (!max_ahead) {
                filp->f_raend  = index + filp->f_ralen;
                filp->f_rawin += filp->f_ralen;
            }
        }
    }
/*
 * The current page is not locked.
 * If we were reading ahead and,
 * if the current max read ahead size is not zero and,
 * if the current position is inside the last read-ahead IO request,
 *   it is the moment to try to read ahead asynchronously.
 * We will later force unplug device in order to force asynchronous read IO.
 */
    else if (reada_ok && filp->f_ramax && raend >= 1 &&
          index <= raend && index + filp->f_ralen >= raend) {
/*
 * Add ONE page to max_ahead in order to try to have about the same IO max size
 * as synchronous read-ahead (MAX_READAHEAD + 1)*PAGE_CACHE_SIZE.
 * Compute the position of the last page we have tried to read in order to
 * begin to read ahead just at the next page.
 */
        raend -= 1;
        if (raend < end_index)
            max_ahead = filp->f_ramax + 1;

        if (max_ahead) {
            filp->f_rawin = filp->f_ralen;
            filp->f_ralen = 0;
            reada_ok      = 2;
        }
    }
/*
 * Try to read ahead pages.
 * We hope that ll_rw_blk() plug/unplug, coalescence, requests sort and the
 * scheduler, will work enough for us to avoid too bad actuals IO requests.
 */
    ahead = 0;
    while (ahead < max_ahead) {
        ahead ++;
        if ((raend + ahead) >= end_index)
            break;
        if (page_cache_read(filp, raend + ahead) < 0)
            break;
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * If we tried to read ahead some pages,
 * If we tried to read ahead asynchronously,
 *   Try to force unplug of the device in order to start an asynchronous
 *   read IO request.
 * Update the read-ahead context.
 * Store the length of the current read-ahead window.
 * Double the current max read ahead size.
 *   That heuristic avoid to do some large IO for files that are not really
 *   accessed sequentially.
 */
	if (ahead) {
		if (reada_ok == 2) {
			run_task_queue(&tq_disk);
		}

		filp->f_ralen += ahead;
		filp->f_rawin += filp->f_ralen;
		filp->f_raend = raend + ahead + 1;

		filp->f_ramax += filp->f_ramax;

		if (filp->f_ramax > max_readahead)
			filp->f_ramax = max_readahead;

		/*
		 * Move the pages that have already been passed
		 * to the inactive list.
		 */
		drop_behind(filp, index);

#ifdef PROFILE_READAHEAD
		profile_readahead((reada_ok == 2), filp);
#endif
	}

	return;
}
```

参数 reada_ok 表示目标页面 page 是否在原来的预读窗口之内，这是在 do_generic_file_read()开头时就计算好了的。

首先要根据具体情况确定一个合适的预读量 max_ahead，这个预读量最初时假定为 0，然后根据具体情况加以修正。怎样修正呢？主要取决于当前页面 page 是否已经锁上，也就是对这个页面中记录块的设备层读入请求是否已经发出。如果已经发出，而当前页面又在先前的预读窗口之内，并且原来的预读窗口中已经包含了对当前页面以后若干页面的预读，那就保持 max_ahead 为 0，最后无功而返。这种情况下 generic_file_readahead()只是作了一个简单的检测而并不是真正进行预读。

如果当前页面虽然已经被锁住，也就是说已经交付设备驱动层加以读入，但是原来并没有预读（flle

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构中的 f_ralen 为 0），或者当前页面已是预读窗口中的最后一个页面或者超出了预读窗口的上沿（index>=raend），或者当前页面在预读窗口的下沿以下，那就说明要另起一个预读窗口了。这个预读窗口的大小首先取决于 file 结构中的 f_ramax，这最初是在 do_generic_file_read()中的 for 循环开始之前与 f_raend 和 f_rawin 一起计算好了的；如果在 for 循环中已经调用过 generic_file_readahead()则由上一次调用遗留下来。根据具体情况的不同，这个数值仍有可能为 0，那就表示不要预读。不过，如果看一下 do_generic_file_read()中的第 1050～1064 行，就可以发现一般情况下这个数值都不会是 0。

如果当前页面没有被锁住，设备驱动层的读入请求尚未发出呢？与前一种情况正好相反。在前一种情况中，是在如果原来没有预读窗口，或者当前页面落在原有窗口之外时才预读。而现在则是如果原来就有预读窗口（read_ok 非 0），并且当前页面落在原有窗口之内时才预读（见 934 行和 835 行）。此时的预读量 max_ahead 也来自 file 结构中的 f_ramax，但是要增加一个页面，因为对当前页面的读入也作为“预读”处理了。如果比较一下在两种情况下对 filp->f_ralen 的初始化，就可以看到在前一种情况下将其设置成 1，因为对当前页面的读入已经在进行中，而在后一种情况下将其设置成 0，因为对当前页面的读入尚未开始。此外，在 949 行还将 reada_ok 设置成 2，表示此时的预读为“异步预读”。其区别在后面会看到。

确定了合适的预读量以后，就开始通过一个 while 循环依次启动对各个页面的读入。函数 page_cache_read()的代码在同一文件 filemap.c 中：

==================== mm/filemap.c 545 577 ====================
[sys_read()>generic_file_read()>do_generic_file_read()>generic_file_readahead()>page_cache_read()]

```
/*
 * This adds the requested page to the page cache if it isn't already there,
 * and schedules an I/O to read in its contents from disk.
 */
static inline int page_cache_read(struct file * file, unsigned long offset)
{
	struct inode *inode = file->f_dentry->d_inode;
	struct address_space *mapping = inode->i_mapping;
	struct page **hash = page_hash(mapping, offset);
	struct page *page;

	spin_lock(&pagecache_lock);
	page = __find_page_nolock(mapping, offset, *hash);
	spin_unlock(&pagecache_lock);
	if (page)
		return 0;

	page = page_cache_alloc();
	if (!page)
		return -ENOMEM;

	if (!add_to_page_cache_unique(page, mapping, offset, hash)) {
		int error = mapping->a_ops->readpage(file, page);
		page_cache_release(page);
		return error;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }
        /*
         * We arrive here in the unlikely event that someone
         * raced with us and added our page to the cache first.
         */
        page_cache_free(page);
        return 0;
}
```

读者对这段代码已经不会感到陌生了，具体的读入由 mapping->a_ops->readpage()启动。对于 Ext2 文件系统这就是 ext2_readpage()，读者已经在前面看过它的代码了。这里要指出的是，如果 find_page_nolock()找到了所需的页面就直接返回 0 而跳过了对页面的读入，但是在上面的 while 循环中却还是将其看成已经进行了预读。当然，这么一来这个已经缓冲的页面可能与设备上不一致，但是也没有什么关系，因为当读这个页面时如果发现仍不一致就会在 do_generic_file_read()中转入 page_not_up_to_date 处加以处理。

只要不在中途到达了文件的终点（end_index 为最后一个逻辑记录块的序号），while 循环就要到完成了由 max_ahead 决定的预读量才会结束。注意，在 ext2_readpage()中只是通过 ll_rw_block()发出了对各个记录块的读入请求，而真正的读入是通过 DMA 进行的，当前进程并不停下来等待其完成。从这个角度讲，所有的磁盘读 / 写其实都是“异步”的，而预读之所以分为“同步”和“异步”，其区别在于第 978 行对 run_task_queue()的调用，即抽出时间做点别的什么，我们在“设备驱动”一章中还会回到这个话题。此外，除对预读窗口的更新外，还将 file 结构中的最大预读量 f_ramax 加倍（见 985 行），只是这个数值不能超过由 get_max_readahead()取得的对于文件所在设备的最大预读量 max_readahead。所以，在同一个预读上下文中，随着预读次数的增加，预读量通常会愈来愈大，直至达到 max_readahead。然后，每当预读上下文改变时，也就是通过 lseek()将当前位置改变到当前预读窗口之外以后的第一次读操作时，就会重新开始一个新的上下文而又开始积累最大预读量。

至于 profile_readahead()，那只是用于统计信息的收集，此处我们就对此不感兴趣了。

5.7 其它文件操作

系统调用 open()、close()和 write()无疑是最基本、最重要、而且也最复杂的文件操作。除此以外，还有许多用于文件操作或与文件操作有关的系统调用。尽管这些系统调用相比之下只是辅助性的，但是在不同的应用中分别起着很重要的作用，限于本书的篇幅，我们不可能对所有这些系统调用都一一列举并加以介绍。读者可以在第 3 章的系统调用函数跳转表中找到与所有这些系统调用对应的内核函数，对于这些（未必是全部）系统调用的作用与运用可以参考关于 Unix/Linux 程序设计的专著。至于实现这些系统调用的代码，则大多数要由读者自己下功夫去阅读了。我们在这里选择其要者介绍几个系统调用的实现。

先看 lseek()，这个系统调用的功能和实现虽然简单但却很重要。内核中实现这个系统调用的函数是 sys_lseek()，其代码在 fs/read_write.c 中：

==================== fs/read_write.c 64 83 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
64   asmlinkage off_t sys_lseek(unsigned int fd, off_t offset, unsigned int origin)
65   {
66        off_t retval;
67        struct file * file;
68
69        retval = -EBADF;
70        file = fget(fd);
71        if (!file)
72             goto bad;
73        retval = -EINVAL;
74        if (origin <= 2) {
75             loff_t res = llseek(file, offset, origin);
76             retval = res;
77             if (res != (loff_t)retval)
78                  retval = -EOVERFLOW;   /* LFS: should only happen on 32 bit platforms */
79        }
80        fput(file);
81   bad:
82        return retval;
83   }
```

参数 origin 的值只能是 0、1 或 2，表示 offset 的起点，本书的读者应该是早就知道的。这段代码中的第 76 行和第 77 行可能会给读者带来一些困惑，既然把 res 的值赋给 retval，怎么二者又可能不相等呢？这是因为二者的类型不同。变量 res 的类型为 loff_t，实际上是 64 位整数；而 retval 的类型为 off_t，是 32 位整数。有关的定义见 include/linux/types.h：

==================== include/linux/types.h 18 18 ====================

```
typedef __kernel_off_t      off_t;
```

==================== include/linux/types.h 45 45 ====================

```
typedef __kernel_loff_t     loff_t;
```

数据类型__kernel_off_t 和__kernel_loff_t 定义则见 include/asm-i386/posix_types.h：

==================== include/asm-i386/posix_types.h 14 14 ====================

```
typedef long       __kernel_off_t;
```

==================== include/asm-i386/posix_types.h 36 36 ====================

```
typedef long long __kernel_loff_t;
```

将 64 位的数值赋给 32 位的变量，然后检查二者是否相等，实际上就是检查这个 32 位数值是否溢出。所以，系统调用 lseek()在 32 位系统结构中只适用于文件大小不超过 4GB 的文件系统。为了突破这个限制，Linux 另外提供了一个系统调用 llseek()，使得在 32 位系统结构中也可以处理大于 4GB 的文件，其代码就在同一文件中，我们就不看了。不过，二者的主体都是 llseek()。函数 llseek()的代码也在同一文件 fs/read_write.c 中：

==================== fs/read_write.c 50 62 ====================

[sys_lseek()>llseek()]

```
static inline loff_t llseek(struct file *file, loff_t offset, int origin)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    loff_t (*fn)(struct file *, loff_t, int);
    loff_t retval;

    fn = default_llseek;
    if (file->f_op && file->f_op->llseek)
        fn = file->f_op->llseek;
    lock_kernel();
    retval = fn(file, offset, origin);
    unlock_kernel();
    return retval;
}
```

注意，这个函数返回值的类型是 64 位的 loff_t，参数 offset 的类型也是 loff_t。

具体的文件系统可以通过 file_operations 结构中的函数指针 llseek 提供相应的函数，以实现这个操作，如果不提供就等于默认采用 default_llseek()。就 Ext2 文件系统而言，它所提供的函数是 ext2_file_lseek()，其代码在 fslext2/file.c 中：

==================== fs/ext2/file.c 39 68 ====================

[sys_lseek()>llseek()>ext2_file_lseek()]

```
/*
 * Make sure the offset never goes beyond the 32-bit mark..
 */
static loff_t ext2_file_lseek(
    struct file *file,
    loff_t offset,
    int origin)
{
    struct inode *inode = file->f_dentry->d_inode;

    switch (origin) {
        case 2:
            offset += inode->i_size;
            break;
        case 1:
            offset += file->f_pos;
    }
    if (offset<0)
        return -EINVAL;
    if (((unsigned long long) offset >> 32) != 0) {
        if (offset > ext2_max_sizes[EXT2_BLOCK_SIZE_BITS(inode->i_sb)])
            return -EINVAL;
    }
    if (offset != file->f_pos) {
        file->f_pos = offset;
        file->f_reada = 0;
        file->f_version = ++event;
    }
    return offset;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
68  }
```

作为参数传递下来的 offset 可以是负值，但根据起始点 origion 加以换算以后就不容许负值了。代码中的第 58、59 行也是对 offset 取值范围的检查。位移 offset 可以超过文件的当前大小，但是不能超过文件大小的上限。这个上限来自两个方面，其一是不能超过 32 位整数的容量，这就是第 58 行所检查的目标；其二是不能超过前一节中所讲 Ext2 文件系统中记录块映射机制的总容量。数组 ext2_max_sizes[]及有关的定义都在 fs/ext2/file.c 中：

==================== fs/ext2/file.c 28 37 ====================

```
#define EXT2_MAX_SIZE(bits)                                    \
     (((EXT2_NDIR_BLOCKS + (1LL << (bits - 2)) +               \
        (1LL << (bits - 2)) * (1LL << (bits - 2)) +            \
        (1LL << (bits - 2)) * (1LL << (bits - 2)) * (1LL << (bits - 2))) * \
       (1LL << bits)) - 1)

static long long ext2_max_sizes[] = {
0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
EXT2_MAX_SIZE(10), EXT2_MAX_SIZE(11), EXT2_MAX_SIZE(12), EXT2_MAX_SIZE(13)
};
```

Ext2 记录块映射机制的总容量取决于记录块大小。当记录块大小为 1K 字节，即 2^{10} 时，这个总容量的大小为 EXT2_MAX_SIZE(10)，由直接、间址、二重间址和三重间址四个部分相加而得。

除将 file 结构中的 f_pos 设置成 offset 的值以外，还将这结构中的 f_reada 设置成 0，因为既然当前位置变了，原来的“预读”上下文就废弃了，到下一次读文件时自会建立起新的预读上下文。此外，event 是系统中的一个全局量，file 结构中的 f_version 字段就以 event 的当前值作为文件读写上下文的“版本号”。

如果孤立地看 ext2_file_lseek()的代码，那么似乎就这么一些了。可是，如果把它与写文件操作的代码结合起来看，里面还隐藏着 lseek()的一个重要性质，那就是可以在文件中创造出“空洞”。原因在于对 offset 的检验只限于文件大小的上限，而并不受文件当前大小的限制。举例来说，假设文件的当前大小是 1KB，而通过 lseek()把“当前位置”移到 9KB 的位置上，然后往文件中写 1 个字节，这么一来，文件的大小变成了 9KB 加 1 个字节，而其中的 8K 字节实际上并没有物理记录块，因此成了空洞。空洞内的逻辑记录块要到往里写的时候才会填上，也就是为之分配物理记录块，在此之前若从空洞中读则为全 0。有些应用软件的程序设计利用了 lseek()的这个性质。不过应指出，应用软件不能用这个方法来“圈地”，因为空洞是没有物理记录块的，并且不保证当往空洞中写时一定能分配到物理记录块，能否分配到记录块要视当时设备上是否尚有空闲记录块而定。

在本节中要看的第二个系统调用是 dup()，用来“复制”一个打开文件号，使新的打开文件号也代表原已存在的文件操作上下文。这个系统调用虽然简单，但是却在 Unix/Linux 系统的运行中扮演着很重要的角色。在内核里面，这个系统调用是由 sys_dup()实现的，其代码在 fs/fcntl.c 中：

==================== fs/fcntl.c 187 195 ====================

```
asmlinkage long sys_dup(unsigned int fildes)
{
     int ret = -EBADF;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct file * file = fget(fildes);

    if (file)
        ret = dupfd(file, 0);
    return ret;
}
```

参数 fildes 为需要“复制”的打开文件号，操作的主体为 dupfd()，其代码中在同一文件中：

==================== fs/fcntl.c 116 131 ====================

[sys_dup()>dupfd()]

```
static int dupfd(struct file *file, int start)
{
    struct files_struct * files = current->files;
    int ret;

    ret = locate_fd(files, file, start);
    if (ret < 0)
        goto out_putf;
    allocate_fd(files, file, ret);
    return ret;

out_putf:
    write_unlock(&files->file_lock);
    fput(file);
    return ret;
}
```

系统调用 dup()的目的是在当前进程的 files_struct 结构内的数组中将一个已打开文件的 file 结构指针复制到另一个原来空闲的位置上，使这个新的“已打开文件”也指向同一个 file 结构。这里的参数 start 表示从数组中的什么位置（下标）开始寻找空闲的数组元素。从 sys_dup()的代码中可以看到传下来的实际参数值是 0。所以 inline 函数 locate_fd()从打开文件号 0 所对应的元素开始寻找，必要时还可以扩充数组的容量。这个函数的代码读者已经在以前看到过，这里就不重复了。找到了空闲的打开文件号以后，就通过 allocate_fd()将作为参数传下来的 file 结构指针“安装”在这个打开文件号所对应的位置上，其代码也在同一个文件 fs/fcntl.c 中：

==================== fs/fcntl.c 107 114 ====================

[sys_dup()>dupfd()>allocate_fd()]

```
static inline void allocate_fd(struct files_struct *files,
                    struct file *file, int fd)
{
    FD_SET(fd, files->open_fds);
    FD_CLR(fd, files->close_on_exec);
    write_unlock(&files->file_lock);
    fd_install(fd, file);
}
```

至于 fd_install()的代码，读者已在 sys_open()中阅读过了。操作完成以后，这个新的打开文件号也

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

就代表着原来由 sys_dup()中的 fildes 所代表的那个文件了。

看似简单的这么一个操作，却起着十分重要的作用，Unix/Linux 各种 shell 的重定向机制就是建立在这个系统调用的基础上的。我们通过实例来看看这具体是怎样实现的。先看这么一条 shell 命令:“ echo what is dup?”。这条命令要求 shell 进程执行一个可执行文件 echo，参数为“ what is dup?”。接收到这条命令，shell 进程先找到可执行文件 bin/echo，然后 fork()出一个子进程让它执行 bin/echo，并将参数传递给它，这个子进程从 shell 继承了标准输入、标准输出以及标准出错信息三个通道，即打开文件号为 0、1、2 的三个已打开文件。至于这个可执行文件本身所规定的操作则是很简单的，就是把参数“ what is dup?”写到标准输出文件，通常就是显示屏上。现在把命令行改成“ echo what is dup? > foo”，要求在执行时将输出“重定向”到一个磁盘文件 foo 中去。对此，shell 进程大体上将执行以下的操作序列，我们假定在此之前该 shell 进程只有三个已打开文件，即打开文件号为 0、1、2 的三个标准输入 / 输出文件：

(1) 打开或创建磁盘文件 foo，并截去 foo 中原有的内容，其打开文件号为 3。
(2) 通过 dup()复制已打开文件 stdout，也就是将打开文件号为 1 处的 file 结构指针复制到打开文件号为 4 处，目的是将 stdout 的 file 指针暂时保存一下。
(3) 关闭 stdout，即 1 号已打开文件，由于它的 file 结构指针已经被复制到打开文件号为 4 处，这个文件（显示屏）实际上并没有最终地关闭，只是把 stdout 的位置腾了出来。
(4) 通过 dup()，复制 3 号已打开文件，由于已打开文件 1 号的位置已经空闲，所以 stdout 位置上的 file 结构指针也就指向了磁盘文件 foo。
(5) 通过系统调用 fork()和 exec()创建子进程并让子进程执行 echo，子进程在执行 echo 前夕将已打开文件 3 号和 4 号关闭而只剩下 0、1、2 三个已打开文件，但是，此时的 stdout，即 1 号已打开文件实际上已指向磁盘文件 foo 而不是显示屏，所以当 echo 将输出往 stdout 写时就写进了文件 foo。
(6) 至于 shell 进程本身，则关闭指向 foo 的 1 号和 3 号已打开文件，并且通过系统调用 dup()和 close()将原来指向显示屏的 file 结构指针恢复到 stdout 位置上，这样 shell 进程就恢复了开始时的三个标准已打开文件。

由此可见，可执行程序 echo 其实并不知道它的标准输出文件 stdout 实际上通向何方，进程与实际输出文件（设备）的结合是在运行时由其父进程“包办”的。

对于 stdin 和 stderr 也是同样。这样就简化了对 echo 的程序设计。因为在程序设计时只要跟三个逻辑上存在的文件打交道就可以了。熟悉“面向对象程序设计”的读者大概会联想到“多态（polymorphism）”和“重载（overload）”这些概念，从而觉得这似乎也没有什么特别高明之处，但是请注意，这个机制的设计与实现是在 30 年以前！

在 dup()的基础上，后来又增设了一个系统调用 dup2()，意为“ dup to”，不同之处在于这个系统调用多一个参数，即作为目标的打开文件号，也就是说，将一个已打开文件复制到指定的位置上。

再来看系统调用 ioctl()，这个系统调用通过一个参数来间接地给出具体的操作命令，所以可以认为是对常规系统调用界面的扩充。其作用就好像是“补遗”或“其它”，凡是操作比较细小，不适合为之专门设置一个系统调用或用去一个系统调用号的，就可以归入这个系统调用。不仅如此，在开发基于 Linux 内核的应用而需要对内核加以扩充（通常是对特殊设备的驱动）时，也常常通过增设新的 ioctl()

操作命令的办法来实现。特别是当这些扩充不能很自然地落入打开、关闭、读、写这些标准的文件操作界面时，ioctl()更是扮演着重要的角色。内核中实现 ioctl()的是 sys_ioctl()，其代码在 fs/ioctl.c 中：

==================== fs/ioctl.c 48 115 ====================

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
	struct file * filp;
	unsigned int flag;
	int on, error = -EBADF;

	filp = fget(fd);
	if (!filp)
		goto out;
	error = 0;
	lock_kernel();
	switch (cmd) {
		case FIOCLEX:
			set_close_on_exec(fd, 1);
			break;

		case FIONCLEX:
			set_close_on_exec(fd, 0);
			break;

		case FIONBIO:
			if ((error = get_user(on, (int *)arg)) != 0)
				break;
			flag = O_NONBLOCK;
#ifdef __sparc__
			/* SunOS compatibility item. */
			if(O_NONBLOCK != O_NDELAY)
				flag |= O_NDELAY;
#endif
			if (on)
				filp->f_flags |= flag;
			else
				filp->f_flags &= ~flag;
			break;

		case FIOASYNC:
			if ((error = get_user(on, (int *)arg)) != 0)
				break;
			flag = on ? FASYNC : 0;

			/* Did FASYNC state change ? */
			if ((flag ^ filp->f_flags) & FASYNC) {
				if (filp->f_op && filp->f_op->fasync)
					error = filp->f_op->fasync(fd, filp, on);
				else error = -ENOTTY;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            }
            if (error != 0)
                break;

            if (on)
                filp->f_flags |= FASYNC;
            else
                filp->f_flags &= ~FASYNC;
            break;

        default:
            error = -ENOTTY;
            if (S_ISREG(filp->f_dentry->d_inode->i_mode))
                error = file_ioctl(filp, cmd, arg);
            else if (filp->f_op && filp->f_op->ioctl)
                error = filp->f_op->ioctl(filp->f_dentry->d_inode, filp, cmd, arg);
    }
    unlock_kernel();
    fput(filp);

out:
    return error;
}
```

参数 fd 为目标文件的打开文件号，cmd 则为具体的操作命令代码。另一个参数 arg 可以用作对具体操作命令的参数，初看之下似乎只能传递一个整型参数对于很多操作是不够的，因而限制了 ioctl() 对内核文件 / 设备操作的扩充能力。其实不然，在需要使用更多参数时可以把这些参数“封装”在一个数据结构中，然后把 arg 用作指向该数据结构的指针，所以，实际上传递参数的能力几乎是不受限制的。

在 include/linux/ioctl.h 中定义了一些命令代码。这些代码的数值大致上是从 0x5401～0x545F，也就是说只使用了两个低字节，并且其中较高的字节都是 0x54，正好是字符“T”的 ASCII 代码。由于参数 cmd 的类型是 32 位无符号整数，其扩充空间是相当大的。

但是，从另一个角度看，由于不同的人在不同的应用中都有可能要通过 ioctl()扩充内核，如何保证命令代码的惟一性而同时又遵循一种统一的格式就成为一个问题，为了这个目的，GNU 建议将 32 位的命令代码 cmd 划分成四个位段（如图 5.8 所示）。

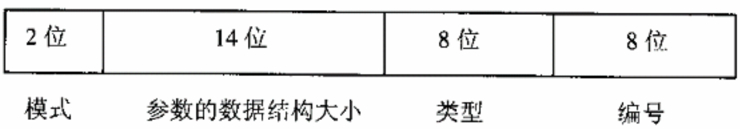


图 5.8　ioctl 命令代码的分段组成

对这些位段的定义与操作，在 include/asm-i386/ioctl.h 中给出了一些定义：

==================== include/asm-i386/ioctl.h 9 59 ====================

```
/* ioctl command encoding: 32 bits total, command in lower 16 bits,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * size of the parameter structure in the lower 14 bits of the
 * upper 16 bits.
 * Encoding the size of the parameter structure in the ioctl request
 * is useful for catching programs compiled with old versions
 * and to avoid overwriting user space outside the user buffer area.
 * The highest 2 bits are reserved for indicating the ``access mode''.
 * NOTE: This limits the max parameter size to 16kB -1 !
 */

/*
 * The following is for compatibility across the various Linux
 * platforms.  The i386 ioctl numbering scheme doesn't really enforce
 * a type field.  De facto, however, the top 8 bits of the lower 16
 * bits are indeed used as a type field, so we might just as well make
 * this explicit here.  Please be sure to use the decoding macros
 * below from now on.
 */
#define _IOC_NRBITS     8
#define _IOC_TYPEBITS   8
#define _IOC_SIZEBITS   14
#define _IOC_DIRBITS    2

#define _IOC_NRMASK     ((1 << _IOC_NRBITS)-1)
#define _IOC_TYPEMASK   ((1 << _IOC_TYPEBITS)-1)
#define _IOC_SIZEMASK   ((1 << _IOC_SIZEBITS)-1)
#define _IOC_DIRMASK    ((1 << _IOC_DIRBITS)-1)

#define _IOC_NRSHIFT    0
#define _IOC_TYPESHIFT  (_IOC_NRSHIFT+_IOC_NRBITS)
#define _IOC_SIZESHIFT  (_IOC_TYPESHIFT+_IOC_TYPEBITS)
#define _IOC_DIRSHIFT   (_IOC_SIZESHIFT+_IOC_SIZEBITS)

/*
 * Direction bits.
 */
#define _IOC_NONE  0U
#define _IOC_WRITE 1U
#define _IOC_READ  2U

#define _IOC(dir,type,nr,size) \
     (((dir)  << _IOC_DIRSHIFT) | \
      ((type) << _IOC_TYPESHIFT) | \
      ((nr)   << _IOC_NRSHIFT) | \
      ((size) << _IOC_SIZESHIFT))

/* used to create numbers */
#define _IO(type,nr)        _IOC(_IOC_NONE,(type),(nr),0)
#define _IOR(type,nr,size)  _IOC(_IOC_READ,(type),(nr),sizeof(size))
#define _IOW(type,nr,size)  _IOC(_IOC_WRITE,(type),(nr),sizeof(size))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define _IOWR(type,nr,size) _IOC(_IOC_READ|_IOC_WRITE,(type),(nr),sizeof(size))
```

例如，Linux 内核中有对网上电话的支持，在网上电话的驱动程序中需要有个控制收听音量的手段。显然，常规的文件操作如 read()、write()、lseek()等都不适用于这个目的，所以就通过扩充 ioctl()的方法来实现这项控制，为此目的需要定义一个命令代码。这个代码的定义在 include/linux/telephony.h 中，我们用它作为一个实例（但是我们在本书中对网上电话的实现本身不感兴趣）：

==================== include/linux/telephony.h 117 117 ====================

```
#define PHONE_PLAY_VOLUME        _IOW ('q', 0x94, int)
```

读者可以根据上面关于_IOW 和各个位段的定义得到命令码 PHONE_PLAY_VOLUME 的数值为 0x40045194。要保证命令码的惟一性，就得保证类型位段或者类型加编号位段取值的惟一性。在从 GNU 下载的 Linux 源代码中有个文件 Documentation/ioctl_number.txt，里面一方面有更具体的说明，另一方面还有个清单，说明类型位段的哪一些数值已经在使用，以及对于给定的类型位段数值哪一些编号已经在使用。需要通过扩充 ioctl()来实现某些设备驱动的读者应参阅这个文件。

至于 sys_ioctl()的代码本身，读者应该没有困难。在 switch 语句内的四种命令码，即 FIOCLEX、FIONCLEX、FIONBIO 和 FIOASYNC，都与具体文件系统无关，只是 VFS 层上的操作。其中 FIOCLEX 将当前进程的 files_sturct 结构中的位图 close_on_exec 内与 fd 相对应的标志位设成 1，使得如果当前进程通过系统调用 exec()执行一个新的可执行程序时就将这个已打开文件自动关闭，而 FIONCLEX 则与之相反。FIONBIO 把对于给定已打开文件的操作设置成“阻塞”或“不阻塞”（blocking/non_blocking）模式。至于 FIOASYNC，则将对此文件的操作设置成“同步”或“异步”模式。通常对已打开文件的操作都是同步和阻塞的，关于不阻塞模式和异步模式可参看下册进程间通信中对“插口”的操作以及设备驱动中的有关内容。

除这几种命令码以外，对常规文件的操作还要由通用的 file_ioctl()再加一层“过滤”，而对设备文件或其它文件（如 FIFO）的处理则直接由具体的 file_operations 结构通过函数指针 ioctl 提供。函数 file_ioctl()的代码在 fs/ioctl.c 中：

==================== fs/ioctl.c 13 45 ====================

[sys_ioctl()>file_ioctl()]

```
static int file_ioctl(struct file *filp,unsigned int cmd,unsigned long arg)
{
	int error;
	int block;
	struct inode * inode = filp->f_dentry->d_inode;

	switch (cmd) {
		case FIBMAP:
		{
			struct address_space *mapping = inode->i_mapping;
			int res;
			/* do we support this mess? */
			if (!mapping->a_ops->bmap)
				return -EINVAL;
			if (!capable(CAP_SYS_RAWIO))
```

```
                    return -EPERM;
                if ((error = get_user(block, (int *) arg)) != 0)
                    return error;

                res = mapping->a_ops->bmap(mapping, block);
                return put_user(res, (int *) arg);
            }
        case FIGETBSZ:
            if (inode->i_sb == NULL)
                return -EBADF;
            return put_user(inode->i_sb->s_blocksize, (int *) arg);
        case FIONREAD:
            return put_user(inode->i_size - filp->f_pos, (int *) arg);
    }
    if (filp->f_op && filp->f_op->ioctl)
        return filp->f_op->ioctl(inode, filp, cmd, arg);
    return -ENOTTY;
}
```

操作命令 FIBMAP 返回文件中给定逻辑块号所对应的物理块号；FIGETBSZ 返回文件所在设备的记录块大小；FIONREAD 则返回文件中从当前读 / 写位置到文件末尾的距离。

除这三种命令以外，就与对设备文件一样，完全取决于具体的文件系统，直接由具体的 file_operations 结构中的函数指针 ioctl 提供。我们以前讲过，每个具体的文件系统都有至少一个 file_operations 数据结构。但是，反过来，每个 file_operations 数据结构却未必都对应着一个不同的文件系统，这里并不是一对一的关系。例如，Ext2 文件系统就有两个这样的数据结构，一个是 ext2_file_operations，另一个是 ext2_dir_operations，分别用于常规文件和目录。以后读者还会看到，对于设备文件和特殊文件，甚至每个文件就可以有其自己的 file_operations 结构。这样，在实现设备驱动程序或特殊文件时，只要单独为其设置一个 file_operations 数据结构，并通过其函数指针 ioctl 提供一个专用的函数，就可以在这个函数中自行定义和实现所需的 ioctl 操作命令了。

我们在这里要看的下一个系统调用是 link()，这个系统调用为已经存在的文件增加一个“别名”。由 link()所建立的是“硬连接”，有别于通过 symlink()建立的“符号连接”。在内核中，link()是由 sys_link()实现的，其代码在 fs/namei.c 中：

==================== fs/namei.c 1579 1632 ====================

```
/*
 * Hardlinks are often used in delicate situations.  We avoid
 * security-related surprises by not following symlinks on the
 * newname.  --KAB
 *
 * We don't follow them on the oldname either to be compatible
 * with linux 2.0, and to avoid hard-linking to directories
 * and other special files.  --ADM
 */
asmlinkage long sys_link(const char * oldname, const char * newname)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
    int error;
    char * from;
    char * to;

    from = getname(oldname);
    if(IS_ERR(from))
        return PTR_ERR(from);
    to = getname(newname);
    error = PTR_ERR(to);
    if (!IS_ERR(to)) {
        struct dentry *new_dentry;
        struct nameidata nd, old_nd;

        error = 0;
        if (path_init(from, LOOKUP_POSITIVE, &old_nd))
            error = path_walk(from, &old_nd);
        if (error)
            goto exit;
        if (path_init(to, LOOKUP_PARENT, &nd))
            error = path_walk(to, &nd);
        if (error)
            goto out;
        error = -EXDEV;
        if (old_nd.mnt != nd.mnt)
            goto out_release;
        new_dentry = lookup_create(&nd, 0);
        error = PTR_ERR(new_dentry);
        if (!IS_ERR(new_dentry)) {
            error = vfs_link(old_nd.dentry, nd.dentry->d_inode, new_dentry);
            dput(new_dentry);
        }
        up(&nd.dentry->d_inode->i_sem);
out_release:
        path_release(&nd);
out:
        path_release(&old_nd);
exit:
        putname(to);
    }
    putname(from);

    return error;
}
```

对于已经阅读了本章前面几节的读者，这里只有一个函数，即 vfs_link()是新的内容，它的代码也在同一文件 fs/namei.c 中：

==================== fs/namei.c 1539 1577 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_link()>vfs_link()]

```
int vfs_link(struct dentry *old_dentry, struct inode *dir, struct dentry *new_dentry)
{
    struct inode *inode;
    int error;

    down(&dir->i_zombie);
    error = -ENOENT;
    inode = old_dentry->d_inode;
    if (!inode)
        goto exit_lock;

    error = may_create(dir, new_dentry);
    if (error)
        goto exit_lock;

    error = -EXDEV;
    if (dir->i_dev != inode->i_dev)
        goto exit_lock;

    /*
     * A link to an append-only or immutable file cannot be created.
     */
    error = -EPERM;
    if (IS_APPEND(inode) || IS_IMMUTABLE(inode))
        goto exit_lock;
    if (!dir->i_op || !dir->i_op->link)
        goto exit_lock;

    DQUOT_INIT(dir);
    lock_kernel();
    error = dir->i_op->link(old_dentry, dir, new_dentry);
    unlock_kernel();

exit_lock:
    up(&dir->i_zombie);
    if (!error)
        inode_dir_notify(dir, DN_CREATE);
    return error;
}
```

这些内容又是读者已经熟悉了的，这里要注意的是对于具有 IS_APPEND 或 IS_IMMUTABLE 属性的文件不允许为之建立别名。具体的连接操作以及这种连接到底意味着什么，则因具体的文件系统而异。所以由具体的文件系统通过其 inode_operations 数据结构中的函数指针 link 来提供，对于 Ext2 文件系统，这个函数是 ext2_link()，它的代码存 fs/ext2/namei.C 中：

==================== fs/ext2/namei.c 663 686 ====================
[sys_link()>vfs_link()>ext2_link()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```c
static int ext2_link (struct dentry * old_dentry,
          struct inode * dir, struct dentry *dentry)
{
     struct inode *inode = old_dentry->d_inode;
     int err;

     if (S_ISDIR(inode->i_mode))
          return -EPERM;

     if (inode->i_nlink >= EXT2_LINK_MAX)
          return -EMLINK;

     err = ext2_add_entry (dir, dentry->d_name.name, dentry->d_name.len,
                    inode);
     if (err)
          return err;

     inode->i_nlink++;
     inode->i_ctime = CURRENT_TIME;
     mark_inode_dirty(inode);
     atomic_inc(&inode->i_count);
     d_instantiate(dentry, inode);
     return 0;
}
```

显然，这个函数在新文件名（别名）所在的目录中创建一个目录项，这个目录项与原来存在的目录项都指向同一个索引节点。这里调用的两个函数 ext2_add_entry()和 d_instantiate()，读者都已经看到过了。函数 d_instantiate()执行完毕以后，inode 结构中的 i_dentry 队列中就多了一个 dentry 结构，即代表着文件的别名的 dentry 结构。同时，这个队列中所有 dentry 结构中的指针 d_inode 都指向这同一个 inode 结构。

由此可见，只要把本章前几节中所涉及的代码读懂，再读其它与文件操作有关的代码就不难了。当然，本书不可能覆盖所有的文件操作或者某一操作的所有细节，许多内容要靠读者自己深入阅读。

如果要问，在文件操作方面还有什么重要的系统调用，那么有两个系统调用是值得一提的，那就是 mknod()和 select()。但是，这两个函数主要用于设备驱动，所以我们把它们放在下册有关设备驱动的章节中。还有一个很重要且有一定难度的系统调用也是与文件操作有关的，那就是 mmap()。这个系统调用将一个已打开文件的内容映射到进程的内存空间，很大程度上是一个存储管理的问题，所以我们已把它放在第 2 章，读者可以在读完本章后回过去阅读。

5.8 特殊文件系统/proc

早期的 Unix 在设备文件目录/dev 下设置了一个特殊文件，称为/dev/mem。通过这个文件可以读／写系统的整个物理内存，而物理内存的地址就用作读／写时文件内部的位移量。这个特殊文件同样适用于 read()、write()、lseek()等常规的文件操作，从而提供了一个在内核外部动态地读／写包括内核映

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

象和内核中各个数据结构以及堆栈内容的手段。这个手段既可用于收集状态信息和统计信息，也可用于程序调试，还可以动态地给内核“打补丁”或改变一些数据结构或变量的内容。采用虚存以后，Unix又增加了一个特殊文件/dev/kmem，对应于系统的整个虚存空间。这两个特殊文件的作用和表现出来的重要性促使人们对其功能加以进一步的拓展，在系统中增设了一个/proc 目录，每当创建一个进程时就以其 pid 为文件名在这个目录下建立起一个特殊文件，使得通过这个文件就可以读 / 写相应进程的用户空间。而当进程 exit()时则将此文件删去。显然，目录/proc 的名称就是这样来的。

经过多年的发展，/proc 成了一个特殊的文件系统，文件系统的类型就叫 proc，其安装点则一般都固定为/proc，所以称其为 proc 文件系统，有时也（非正式地）称之为/proc 文件系统。这个文件系统中所有的文件都是特殊文件，这些文件的内容都不存在于任何设备上，而是在读 / 写的时候才根据系统中的有关信息生成出来，或者映射到系统中的有关变量或数据结构。所以又称为“伪文件系统”。同时，这个子系统中的内容也已扩展到了足以覆盖系统的几乎所有方面，而不再仅仅是关于各个进程的信息。限于篇幅，我们不在这里列出/proc 目录下的内容，建议读者自己用命令“ du -a /proc”看一下。笔者试了一下这个命令，其结果达 1000 行以上！大体上，这个目录下的内容主要包括如下几类：

(1) 系统中的每个进程都有一个以其 pid 为名的子目录，而每个子目录中则包括关于该进程执行的命令行、所有环境变量、cpu 占用时间、内存映射表、已打开文件的文件号以及进程状态等特殊文件。
(2) 系统中各种资源的管理信息，如/proc/slabinfo 就是内存管理中关于各个 slab 缓冲块队列的信息，/proc/swaps 就是关于系统的 swap 设备的信息，/proc/partitions 就是关于各个磁盘分区的信息，等等。
(3) 系统中各种设备的有关信息，如/proc/pci 就是关于系统的 PCI 总线上所有设备的一份清单，等等。
(4) 文件系统的信息，如/proc/mounts 就是系统中已经安装的各个文件系统设备的清单，而/proc/filesystems 则是系统中已经登记的每种文件系统（类型）的清单。
(5) 中断的使用，/proc/interrupts 是一份关于中断源和它们的中断向量编号的清单。
(6) 与动态安装模块有关的信息，/proc/modules 是一份系统中已安装动态模块的清单，而/proc/ksyms 则是内核中供可安装模块动态连接的符号名及其地址的清单。
(7) 与前述/dev/mem 类似的内存访问手段，如/proc/kcore。
(8) 系统的版本号以及其它各种统计与状态信息。

读者可以通过命令“ man proc”，看一下对这些信息的说明。

不仅如此，动态安装模块还可以在/proc 目录下动态地创建文件，并以此作为模块与用户进程间的界面。

由于 proc 文件系统并不物理地存在于任何设备上，它的安装过程是特殊的。对 proc 文件系统不能直接通过 mount()来安装，而要先由系统内核在内核初始化时自动地通过一个函数 kern_mount()安装一次，然后再由处理系统初始化的进程通过 mount()安装，实际上是“重安装”。我们来看有关的代码，首先是 fs/proc/procfs_syms.c 中的 init_proc_fs()，这是在内核初始化时调用的：

==================== fs/proc/procfs_syms.c 23 37 ====================

```
static DECLARE_FSTYPE(proc_fs_type, "proc", proc_read_super, FS_SINGLE);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static int __init init_proc_fs(void)
{
	int err = register_filesystem(&proc_fs_type);
	if (!err) {
		proc_mnt = kern_mount(&proc_fs_type);
		err = PTR_ERR(proc_mnt);
		if (IS_ERR(proc_mnt))
			unregister_filesystem(&proc_fs_type);
		else
			err = 0;
	}
	return err;
}
```

系统在初始化阶段对 proc 文件系统做两件事，一是通过 register_filesystem()向系统登记“ proc”这么一种文件系统；二是通过 kern_mount()将一个具体的 proc 文件系统安装到系统中的/proc 结点上。函数 kern_mount()的代码在 fs/super.c 中：

==================== fs/super.c 970 989 ====================
[init_proc_fs()>kern_mount()]

```
struct vfsmount *kern_mount(struct file_system_type *type)
{
	kdev_t dev = get_unnamed_dev();
	struct super_block *sb;
	struct vfsmount *mnt;
	if (!dev)
		return ERR_PTR(-EMFILE);
	sb = read_super(dev, NULL, type, 0, NULL, 0);
	if (!sb) {
		put_unnamed_dev(dev);
		return ERR_PTR(-EINVAL);
	}
	mnt = add_vfsmnt(NULL, sb->s_root, NULL);
	if (!mnt) {
		kill_super(sb, 0);
		return ERR_PTR(-ENOMEM);
	}
	type->kern_mnt = mnt;
	return mnt;
}
```

每个已安装的文件系统都要有个 super_block 数据结构，proc 文件系统也不例外。由于 super_block 数据结构需要有个设备号来惟一地加以标识，尽管 proc 文件系统并不实际存在于任何设备上，却也得有个“设备号”，所以要通过 get_unnamed_dev()分配一个。这个函数的代码也在 super.c 中：

==================== fs/super.c 757 773 ====================
[init_proc_fs()>kern_mount()>get_unnamed_dev()]

```
/*
 * Unnamed block devices are dummy devices used by virtual
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * filesystems which don't use real block-devices.  -- jrs
 */

static unsigned int unnamed_dev_in_use[256/(8*sizeof(unsigned int))];

kdev_t get_unnamed_dev(void)
{
     int i;

     for (i = 1; i < 256; i++) {
          if (!test_and_set_bit(i,unnamed_dev_in_use))
               return MKDEV(UNNAMED_MAJOR, i);
     }
     return 0;
}
```

这个“设备号”的主设备号为 UNNAMED_MAJOR，定义为 0。

除此之外，kern_mount()中调用的函数读者在“文件系统的安装和拆卸”一节中都已看过了。函数 read_super()（见“文件系统的安装与拆卸”）分配一个空白的 super_block 数据结构，然后通过由具体文件系统的 file_system_type 数据结构中的函数指针 read_super 调用具体的函数来读入超级块。对于 proc 文件系统，这个函数为 proc_read_super()，这是在上面的宏定义 DECLARE_FSTYPE 中定义好的，其代码在 fs/proc/inode.c 中：

==================== fs/proc/inode.c 181 210 ====================
[init_proc_fs()>kern_mount()>read_super()>proc_read_super()]

```
struct super_block *proc_read_super(struct super_block *s,void *data,
                        int silent)
{
     struct inode * root_inode;
     struct task_struct *p;

     s->s_blocksize = 1024;
     s->s_blocksize_bits = 10;
     s->s_magic = PROC_SUPER_MAGIC;
     s->s_op = &proc_sops;
     root_inode = proc_get_inode(s, PROC_ROOT_INO, &proc_root);
     if (!root_inode)
          goto out_no_root;
     /*
      * Fixup the root inode's nlink value
      */
     read_lock(&tasklist_lock);
     for_each_task(p) if (p->pid) root_inode->i_nlink++;
     read_unlock(&tasklist_lock);
     s->s_root = d_alloc_root(root_inode);
     if (!s->s_root)
          goto out_no_root;
     parse_options(data, &root_inode->i_uid, &root_inode->i_gid);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    return s;

out_no_root:
    printk("proc_read_super: get root inode failed\n");
    iput(root_inode);
    return NULL;
}
```

可见，说是“读入超级块”，实际上却是“生成超级块”。还有，super_block 结构中的 super_operations 指针 s_op 被设置成指向 proc_sops，这也是在 fs/proc/inode.c 中定义的：

==================== fs/proc/inode.c 94 99 ====================

```
static struct super_operations proc_sops = {
    read_inode:   proc_read_inode,
    put_inode:    force_delete,
    delete_inode: proc_delete_inode,
    statfs:       proc_statfs,
};
```

读者将会看到，proc 文件系统的 inode 结构也像其 super_block 结构一样，在设备上并没有对应物，而仅仅是在内存中生成的“空中楼阁”。这些函数正是为这些“空中楼阁”服务的。

不仅如此，proc 文件系统中的目录项结构，即 dentry 结构，在设备上也没有对应物，而以内存中的 proc_dir_entry 数据结构来代替，定义于 include/linux/proc_fs.h：

==================== include/linux/proc_fs.h 53 73 ====================

```
struct proc_dir_entry {
    unsigned short low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct inode_operations * proc_iops;
    struct file_operations * proc_fops;
    get_info_t *get_info;
    struct module *owner;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;
    write_proc_t *write_proc;
    atomic_t count;        /* use count */
    int deleted;       /* delete flag */
    kdev_t   rdev;
};
```

显然，这个数据结构中的有些内容本身应该是属于 inode 结构或索引节点的，所以实际上既是创建

dentry 结构的依据，又是 inode 结构中部分信息的来源。如果与 Ext2 文件系统中的 ext2_dir_entry 结构相比，则那是存储在设备上的“目录项”，而 proc_dir_entry 结构只存在于内存中，并且包含了更多的信息。这些 proc_dir_entry 结构多数都是在系统的运行中动态地分配空间而创立的，但是也有一些是静态定义的，其中最重要的就是 proc 文件系统的根节点，即/proc 的目录项 proc_root，定义于 fs/proc/root.c 中：

==================== fs/proc/root.c 96 108 ====================

```
/*
 * This is the root "inode" in the /proc tree..
 */
struct proc_dir_entry proc_root = {
	low_ino: PROC_ROOT_INO,
	namelen: 5,
	name:		"/proc",
	mode:		S_IFDIR | S_IRUGO | S_IXUGO,
	nlink:		2,
	proc_iops:	&proc_root_inode_operations,
	proc_fops:	&proc_root_operations,
	parent:		&proc_root,
};
```

注意，这个数据结构中的指针 proc_iops 指向 proc_root_inode_operations，而 proc_fops 指向 proc_root_operations。还有，结构中的指针 parent 指向其自己，即 proc_root；也就是说，这个节点是一个文件系统的根节点。

回到 proc_read_super()的代码中，数据结构 proc_root 就用来创建根节点/proc 的 inode 结构，其中 PROC_ROOT_INO 的定义在文件 proc_fs.h 中给出：

==================== include/linux/proc_fs.h 22 22 ====================

```
	PROC_ROOT_INO = 1,
```

所以，用于/proc 的 inode 节点号总是 1，而设备上的 1 号索引节点是保留不用的。

函数 proc_get_inode()的代码在 fs/proc/inode.c 中：

==================== fs/proc/inode.c 131 179 ====================

[init_proc_fs()>kern_mount()>read_super()>proc_read_super()>proc_get_inode()]

```
struct inode * proc_get_inode(struct super_block * sb, int ino,
				struct proc_dir_entry * de)
{
	struct inode * inode;

	/*
	 * Increment the use count so the dir entry can't disappear.
	 */
	de_get(de);
#if 1
/* shouldn't ever happen */
if (de && de->deleted)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
printk("proc_iget: using deleted entry %s, count=%d\n", de->name, atomic_read(&de->count));
#endif

    inode = iget(sb, ino);
    if (!inode)
        goto out_fail;

    inode->u.generic_ip = (void *) de;
    if (de) {
        if (de->mode) {
            inode->i_mode = de->mode;
            inode->i_uid = de->uid;
            inode->i_gid = de->gid;
        }
        if (de->size)
            inode->i_size = de->size;
        if (de->nlink)
            inode->i_nlink = de->nlink;
        if (de->owner)
            __MOD_INC_USE_COUNT(de->owner);
        if (S_ISBLK(de->mode)||S_ISCHR(de->mode)||S_ISFIFO(de->mode))
            init_special_inode(inode,de->mode,kdev_t_to_nr(de->rdev));
        else {
            if (de->proc_iops)
                inode->i_op = de->proc_iops;
            if (de->proc_fops)
                inode->i_fop = de->proc_fops;
        }
    }

out:
    return inode;

out_fail:
    de_put(de);
    goto out;
}
```

我们知道，inode 结构包含着一个 union，视具体的文件系统而用作不同的数据结构，例如对于 Ext2 文件系统就用作 ext2_inode_info 结构，在 inode 数据结构的定义中列出了适用于不同文件系统的不同数据结构。如果具体的文件系统不在其列，则将这个 union（的开头 4 个字节）解释为一个指针，这就是 generic_ip。在这里，就将这个指针设置成指向相应的 proc_dir_entry 结构，使其在逻辑上成为 inode 结构的一部分。至于 de_get()，那只是递增数据结构中的使用计数而已，此外，iget()是个 inlinc 函数，读者已经在前几节中看到过了。在这里，由于相应的 inode 结构还不存在，实际上会调用 get_new_inode()分配一个 inode 结构。

创建了 proc 文件系统根节点的 inode 结构以后，还要通过 d_alloc_root()创建其 dentry 结构。这个函数的代码读者已在“文件系统的安装与拆卸”一节中看到过了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里还有个有趣的事，就是对系统中除 0 号进程以外的所有进程都递增该 inode 结构中的 i_nlink 字段。这样，只要这些进程中的任何一个还存在，就不能把这个 inode 结构删除。

回到 kern_mount()，函数 add_vfsmnt()的代码也是读者已经看到过的。但是，要注意这里调用这个函数时的参数。第一个参数 nd 是个 nameidata 结构指针，本来应该指向代表着安装点的 nameidata 结构，从这个结构里就可以得到指向安装点 dentry 结构的指针。可是，这里的调用参数却是 NULL。第二个参数 root 是个 dentry 结构指针，指向待安装文件系统中根目录的 dentry 结构，在这里是 proc 文件系统根节点的 dentry 数据结构。可是，如果指向安装点的指针是 NULL，那怎么安装呢？我们来看 add_vfsmnt()中的主体：

==================== fs/super.c 337 350 ====================

```
	mnt->mnt_root = dget(root);
	mnt->mnt_mountpoint = nd ? dget(nd->dentry) : dget(root);
	mnt->mnt_parent = nd ? mntget(nd->mnt) : mnt;

	if (nd) {
		list_add(&mnt->mnt_child, &nd->mnt->mnt_mounts);
		list_add(&mnt->mnt_clash, &nd->dentry->d_vfsmnt);
	} else {
		INIT_LIST_HEAD(&mnt->mnt_child);
		INIT_LIST_HEAD(&mnt->mnt_clash);
	}
	INIT_LIST_HEAD(&mnt->mnt_mounts);
	list_add(&mnt->mnt_instances, &sb->s_mounts);
	list_add(&mnt->mnt_list, vfsmntlist.prev);
```

可见，在参数 nd 为 NULL 时，安装以后其 vfsmount 结构中的指针 mnt_mountpoint 指向待安装文件系统中根目录的 dentry 结构，即 proc 文件系统根节点的 dentry 结构本身；指针 mnt_parent 则指向这个 vfsmount 结构本身。并且，这个 vfsmount 结构的 mnt_child 和 mnt_clash 两个队列头也空着不用。显然，这个 vfsmount 结构并没有把 proc 文件系统的根节点"安装"到什么地方。可是，回到 kern_mount()的代码中，下面还有一行重要的语句：

==================== fs/super.c 987 987 ====================

```
	type->kern_mnt = mnt;
```

这个语句使 proc 文件系统的 file_system_type 数据结构与上面的 vfsmount 结构挂上了钩，使它的指针 kern_mnt 指向了这个 vfsmount 结构。可是，这并不意味着 path_walk()就能顺着路径名" /proc"找到 proc 文件系统的根节点，因为 path_walk()并不涉及 file_system_type 数据结构。

正因为如此，光是 kern_mount()还不够，还得由系统的初始化进程从内核外部通过系统调用 mount()再安装一次。通常，这个命令行为：

mount -nvt proc /dev/null /proc

就是说，把建立在"空设备" /dev/null 上的 proc 文件系统安装在节点/proc 上。从理论讲也可以把它安装在其它节点上，但实际上总是安装在/proc 上。

前面我们提到过，proc 文件系统的 file_system_type 数据结构中的 FS_SINGLE 标志位为 1，它起着重要的作用。为什么重要呢？因为它使 sys_mount()的主体 do_mount()通过 get_sb_single()，而不是

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

get_sb_bdev()，来取得所安装文件系统的 super_block 数据结构。我们回顾一下 do_mount()中与此有关的片段：

==================== fs/super.c 1371 1377 ====================

```
	/* get superblock, locks mount_sem on success */
	if (fstype->fs_flags & FS_NOMOUNT)
		sb = ERR_PTR(-EINVAL);
	else if (fstype->fs_flags & FS_REQUIRES_DEV)
		sb = get_sb_bdev(fstype, dev_name, flags, data_page);
	else if (fstype->fs_flags & FS_SINGLE)
		sb = get_sb_single(fstype, flags, data_page);
```

我们在“文件系统的安装与拆卸”一章中阅读 do_mount()的代码时跳过了 get_sb_single()，现在要回过来看它的代码了（fs/super.c）。

==================== fs/super.c 870 885 ====================
[sys_mount()>do_mount()>get_sb_single()]

```
static struct super_block *get_sb_single(struct file_system_type *fs_type,
	int flags, void *data)
{
	struct super_block * sb;
	/*
	 * Get the superblock of kernel-wide instance, but
	 * keep the reference to fs_type.
	 */
	down(&mount_sem);
	sb = fs_type->kern_mnt->mnt_sb;
	if (!sb)
		BUG();
	get_filesystem(fs_type);
	do_remount_sb(sb, flags, data);
	return sb;
}
```

代码中通过 file_system_type 结构中的指针 kern_mnt 取得对于所安装文件系统的 vfsmount 结构，从而对其 super_block 结构的访问，而这正是在 kern_mount()中设置好了的。这里还调用了一个函数 do_remount_sb()，其代码也在 fs/super.c 中：

==================== fs/super.c 936 968 ====================
[sys_mount()>do_mount()>get_sb_single()>do_remount_sb()]

```
/*
 * Alters the mount flags of a mounted file system. Only the mount point
 * is used as a reference - file system type and the device are ignored.
 */

static int do_remount_sb(struct super_block *sb, int flags, char *data)
{
	int retval;

```

```
    if (!(flags & MS_RDONLY) && sb->s_dev && is_read_only(sb->s_dev))
        return -EACCES;
        /*flags |= MS_RDONLY;*/
    /* If we are remounting RDONLY, make sure there are no rw files open */
    if ((flags & MS_RDONLY) && !(sb->s_flags & MS_RDONLY))
        if (!fs_may_remount_ro(sb))
            return -EBUSY;
    if (sb->s_op && sb->s_op->remount_fs) {
        lock_super(sb);
        retval = sb->s_op->remount_fs(sb, &flags, data);
        unlock_super(sb);
        if (retval)
            return retval;
    }
    sb->s_flags = (sb->s_flags & ~MS_RMT_MASK) | (flags & MS_RMT_MASK);

    /*
     * We can't invalidate inodes as we can loose data when remounting
     * (someone might manage to alter data while we are waiting in lock_super()
     * or in foo_remount_fs()))
     */

    return 0;
}
```

这个函数对于 proc 文件系统作用不大，因为 proc 并无特殊的 remount_fs 操作。标志位屏蔽模式 MS_RMT_MASK 的定义在 include/linux/fs.h 中：

==================== include/linux/fs.h 110 114 ====================

```
/*
 * Flags that can be altered by MS_REMOUNT
 */
#define MS_RMT_MASK    (MS_RDONLY|MS_NOSUID|MS_NODEV|MS_NOEXEC|\
            MS_SYNCHRONOUS|MS_MANDLOCK|MS_NOATIME|MS_NODIRATIME)
```

经过 do_remount_sb()以后，原来 super_block 结构中的这些标志位就由用户所提供的相应标志位所取代。

取得了 proc 文件系统的 super_block 结构以后，回到 do_mount()的代码中，此后的操作就与普通文件系统的安装无异了。这样，就将 proc 文件系统安装到了节点/proc 上。

前面讲过，整个 proc 文件系统都不存在于设备上，所以不光是它的根节点需要在内存中创造出来，自根节点以下的所有节点全都需要在运行时加以创建，这是由内核在初始化时调用 proc_root_init()完成的，其代码在 fs/proc/ro0t.c 中：

==================== fs/proc/root.c 25 46 ====================

```
void __init proc_root_init(void)
{
    proc_misc_init();
    proc_net = proc_mkdir("net", 0);
```

```
#ifdef CONFIG_SYSVIPC
        proc_mkdir("sysvipc", 0);
#endif
#ifdef CONFIG_SYSCTL
        proc_sys_root = proc_mkdir("sys", 0);
#endif
        proc_root_fs = proc_mkdir("fs", 0);
        proc_root_driver = proc_mkdir("driver", 0);
#if defined(CONFIG_SUN_OPENPROMFS) || defined(CONFIG_SUN_OPENPROMFS_MODULE)
        /* just give it a mountpoint */
        proc_mkdir("openprom", 0);
#endif
        proc_tty_init();
#ifdef CONFIG_PROC_DEVICETREE
        proc_device_tree_init();
#endif
        proc_bus = proc_mkdir("bus", 0);
}
```

首先是直接在/proc 下面的叶节点，即文件节点，这是由 proc_misc_init()创建的，其代码在 fs/proc/proc_misc.c 中：

==================== fs/proc/proc_misc.c 505 582 ====================

[proc_root_init()>proc_misc_init()]

```
void __init proc_misc_init(void)
{
        struct proc_dir_entry *entry;
        static struct {
                char *name;
                int (*read_proc)(char*,char**,off_t,int,int*,void*);
        } *p, simple_ones[] = {
                {"loadavg",     loadavg_read_proc},
                {"uptime",      uptime_read_proc},
                {"meminfo",     meminfo_read_proc},
                {"version",     version_read_proc},
                {"cpuinfo",     cpuinfo_read_proc},
#ifdef CONFIG_PROC_HARDWARE
                {"hardware",    hardware_read_proc},
#endif
#ifdef CONFIG_STRAM_PROC
                {"stram", stram_read_proc},
#endif
#ifdef CONFIG_DEBUG_MALLOC
                {"malloc",      malloc_read_proc},
#endif
#ifdef CONFIG_MODULES
                {"modules",     modules_read_proc},
                {"ksyms", ksyms_read_proc},
#endif
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         {"stat",  kstat_read_proc},
         {"devices",   devices_read_proc},
         {"partitions",    partitions_read_proc},
#if !defined(CONFIG_ARCH_S390)
         {"interrupts",    interrupts_read_proc},
#endif
         {"filesystems",   filesystems_read_proc},
         {"dma",       dma_read_proc},
         {"ioports",   ioports_read_proc},
         {"cmdline",   cmdline_read_proc},
#ifdef CONFIG_SGI_DS1286
         {"rtc",       ds1286_read_proc},
#endif
         {"locks", locks_read_proc},
         {"mounts",    mounts_read_proc},
         {"swaps", swaps_read_proc},
         {"iomem", memory_read_proc},
         {"execdomains",   execdomains_read_proc},
         {NULL,}
    };
    for (p = simple_ones; p->name; p++)
         create_proc_read_entry(p->name, 0, NULL, p->read_proc, NULL);

    /* And now for trickier ones */
    entry = create_proc_entry("kmsg", S_IRUSR, &proc_root);
    if (entry)
         entry->proc_fops = &proc_kmsg_operations;
    proc_root_kcore = create_proc_entry("kcore", S_IRUSR, NULL);
    if (proc_root_kcore) {
         proc_root_kcore->proc_fops = &proc_kcore_operations;
         proc_root_kcore->size =
                   (size_t)high_memory - PAGE_OFFSET + PAGE_SIZE;
    }
    if (prof_shift) {
         entry = create_proc_entry("profile", S_IWUSR | S_IRUGO, NULL);
         if (entry) {
              entry->proc_fops = &proc_profile_operations;
              entry->size = (1+prof_len) * sizeof(unsigned int);
         }
    }
#ifdef __powerpc__
    {
         extern struct file_operations ppc_htab_operations;
         entry = create_proc_entry("ppc_htab", S_IRUGO|S_IWUSR, NULL);
         if (entry)
              entry->proc_fops = &ppc_htab_operations;
    }
#endif
    entry = create_proc_read_entry("slabinfo", S_IWUSR | S_IRUGO, NULL,
```

```
                              slabinfo_read_proc, NULL);
        if (entry)
                entry->write_proc = slabinfo_write_proc;
}
```

这个函数的前半部是一个数据结构数组的定义。这个数组中的每一个元素都将/proc 目录中的一个（文件）节点名与一个函数挂上钩。例如，节点/proc/cpuinfo 就与 cpuinfo_read_proc()挂钩，当一个进程访问这个节点，要读出这个特殊文件的内容时，就由 cpuinfo_read_proc()从内核中收集有关的信息并临时生成该文件的内容。这个数组中所涉及的所有特殊文件都只支持读操作，而不支持其它的文件操作（如写、lseek()等）。看一下这个数组，即 simple_ones[]，读者就可以约略地感受到在/proc 下面的这些特殊文件所提供的信息是何等地充分和多样。所有这些函数都要通过 create_proc_read_entry()为之创建起 proc_dir_entry 结构和 inode 结构，并且与节点/proc 的数据结构挂上钩，这是定义于 include/linux/proc.h 中的一个 inline 函数：

==================== include/linux/proc_fs.h 135 145 ====================
[proc_root_init()>proc_misc_init()>create_proc_read_entry()]

```
extern inline struct proc_dir_entry *create_proc_read_entry(const char *name,
        mode_t mode, struct proc_dir_entry *base,
        read_proc_t *read_proc, void * data)
{
        struct proc_dir_entry *res=create_proc_entry(name,mode,base);
        if (res) {
                res->read_proc=read_proc;
                res->data=data;
        }
        return res;
}
```

对照一下调用时的参数，就可以看到这里除 name 和 read_proc 以外其它参数都是 0 或 NULL，特别地，文件的 mode 为 0。所以这里做的就是通过 create_proc_entry()建立起有关的数据结构并将所创建 proc_dir_entry 结构中的函数指针 read_proc 设置成指向相应的函数。

函数 create_proc_entry()的代码在 fs/proc/generic.c 中：

==================== fs/proc/generic.c 497 535 ====================
[proc_root_init()>proc_misc_init()>create_proc_read_entry()>create_proc_entry()]

```
struct proc_dir_entry *create_proc_entry(const char *name, mode_t mode,
                                         struct proc_dir_entry *parent)
{
        struct proc_dir_entry *ent = NULL;
        const char *fn = name;
        int len;

        if (!parent && xlate_proc_name(name, &parent, &fn) != 0)
                goto out;
        len = strlen(fn);

        ent = kmalloc(sizeof(struct proc_dir_entry) + len + 1, GFP_KERNEL);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
509         if (!ent)
510             goto out;
511         memset(ent, 0, sizeof(struct proc_dir_entry));
512         memcpy(((char *) ent) + sizeof(*ent), fn, len + 1);
513         ent->name = ((char *) ent) + sizeof(*ent);
514         ent->namelen = len;
515
516         if (S_ISDIR(mode)) {
517             if ((mode & S_IALLUGO) == 0)
518             mode |= S_IRUGO | S_IXUGO;
519             ent->proc_fops = &proc_dir_operations;
520             ent->proc_iops = &proc_dir_inode_operations;
521             ent->nlink = 2;
522         } else {
523             if ((mode & S_IFMT) == 0)
524                 mode |= S_IFREG;
525             if ((mode & S_IALLUGO) == 0)
526                 mode |= S_IRUGO;
527             ent->nlink = 1;
528         }
529         ent->mode = mode;
530
531         proc_register(parent, ent);
532
533     out:
534         return ent;
535     }
```

在这个情景中，进入这个函数时的参数 parent 为 NULL，所以在第 505 行调用 xlate_proc_name()，它将作为参数传下来的节点名（如“cpuinfo”）转换成从“/proc”开始的路径名，并且通过副作用返回指向节点 proc 的 proc_dir_entry 结构的指针作为 parent。显然，这个函数的作用在这里是很关键的，其代码在 fs/proc/generic.c 中：

==================== fs/proc/generic.c 161 191 ====================

[proc_root_init()>proc_misc_init()>create_proc_read_entry()>create_proc_entry()>xlate_proc_name()]

```
/*
 * This function parses a name such as "tty/driver/serial", and
 * returns the struct proc_dir_entry for "/proc/tty/driver", and
 * returns "serial" in residual.
 */
static int xlate_proc_name(const char *name,
                struct proc_dir_entry **ret, const char **residual)
{
    const char            *cp = name, *next;
    struct proc_dir_entry   *de;
    int            len;

    de = &proc_root;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
174      while (1) {
175          next = strchr(cp, '/');
176          if (!next)
177              break;
178
179          len = next - cp;
180          for (de = de->subdir; de ; de = de->next) {
181              if (proc_match(len, cp, de))
182                  break;
183          }
184          if (!de)
185              return -ENOENT;
186          cp += len + 1;
187      }
188      *residual = cp;
189      *ret = de;
190      return 0;
191  }
```

这里 proc_root 就是根节点/proc 的 proc_dir_entry 结构，结构中的 subdir 是一个队列。下面读者就会看到，/proc 下所有节点的 proc_dir_entry 结构都在这个队列中。函数 strchr()在字符串 cp 中寻找第一个“/”字符并返回指向该字符的指针。函数中的 while 循环逐字节地检查作为相对路径名的字符串，在/proc 下面的子目录队列中寻求匹配。对于字符串“cpuinfo”来说，由于字符串中并无“/”字符存在，因而 strchr()返回 NULL 而经由第 177 行的 break 语句结束 while 循环。所以，对于相对路径名“cpuinfo”而言，这个函数返回 0，并且在返回 create_proc_entry()后使 parent 指向/proc 的 proc_dir_entry 结构，而 fn 则保持不变。这么一来，在为节点“cpuinfo”分配 proc_dir_entry 结构并加以初始化以后，当调用 proc_register()时，parent 就一定指向/proc 或其它给定父节点的 proc_dir_entry 结构。

函数 proc_register()将一个新节点的 proc_dir_entry 结构“登记”（即挂入）到父节点的 proc_dir_entry 结构内的 subdir 队列中，它的源代码在同一文件 fs/proc/generic.c 中：

==================== fs/proc/generic.c 350 375 ====================

[proc_root_init()>proc_misc_init()>create_proc_read_entry()>create_proc_entry()>proc_register()]

```
static int proc_register(struct proc_dir_entry * dir, struct proc_dir_entry * dp)
{
    int  i;

    i = make_inode_number();
    if (i < 0)
        return -EAGAIN;
    dp->low_ino = i;
    dp->next = dir->subdir;
    dp->parent = dir;
    dir->subdir = dp;
    if (S_ISDIR(dp->mode)) {
        if (dp->proc_iops == NULL) {
            dp->proc_fops = &proc_dir_operations;
            dp->proc_iops = &proc_dir_inode_operations;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }
        dir->nlink++;
    } else if (S_ISLNK(dp->mode)) {
        if (dp->proc_iops == NULL)
            dp->proc_iops = &proc_link_inode_operations;
    } else if (S_ISREG(dp->mode)) {
        if (dp->proc_fops == NULL)
            dp->proc_fops = &proc_file_operations;
    }
    return 0;
}
```

参数 dir 指向父节点，dp 则指向要登记的节点。如前所述，/proc 及其下属的所有节点在设备上都没有对应的索引节点，但是在内存中却都有 inode 数据结构。既然有 inode 结构，就要有索引节点号。proc 文件系统的根节点即/proc 节点的索引节点号为 1，可是其它节点呢？这就要通过 make_inode_number()予以分配了，此函数的代码也在文件 fs/proc/generic.c 中：

==================== fs/proc/generic.c 193 202 ====================
[proc_root_init()>proc_misc_init()>create_proc_read_entry()>create_proc_entry()>proc_register()>make_inode_number()]

```
static unsigned char proc_alloc_map[PROC_NDYNAMIC / 8];

static int make_inode_number(void)
{
    int i = find_first_zero_bit((void *) proc_alloc_map, PROC_NDYNAMIC);
    if (i<0 || i>=PROC_NDYNAMIC)
        return -1;
    set_bit(i, (void *) proc_alloc_map);
    return PROC_DYNAMIC_FIRST + i;
}
```

代码中用到的几个常量在文件 include/linux/proc_fs.h 中给出：

==================== include/linux/proc_fs.h 25 28 ====================

```
/* Finally, the dynamically allocatable proc entries are reserved: */

#define PROC_DYNAMIC_FIRST 4096
#define PROC_NDYNAMIC      4096
```

可见，这些节点的索引节点号都在 4096～8192 范围内。索引节点号并不需要在整个系统的范围中保持惟一，而只要在同一设备的范围中惟一就可以了。当然，/proc 下面的节点都不属于任何设备，但是也有个设备，所以这些索引节点号也因此而不会与任何一个具体设备上的索引节点号冲突。

从代码中可以看出，proc 文件系统有两个 file_operations 数据结构，即 proc_dir_operations 和 proc_file_operations，以及两个 inode_operations 数据结构，即 proc_dir_inode_operations 和 proc_link_inode_operations，视具体节点的类型加以设置。例如，下面讲到的/proc/self 就是一个连接节点，所以其 proc_iops 指向 proc_link_operations。

此外，proc_register()中的代码就没有什么需要特别加以说明的了。不过，从“cpuinfo”这个例子可以看出，所谓 subdir 队列并非“子目录的队列”，而是“下属节点的目录项的队列”。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

回到 proc_misc_init()的代码中，除数组 simple_ones[]中的节点外，还有“ kmsg”、“ kcore”以及“ profile ”三个节点。由于这些特殊文件的访问权限有所不同，例如“ kmsg” 和“ kcore” 的 mode 都是 S_IRUSR，也就是只有文件主即特权用户才有读访问权，所以不能一律套用 create_proc_read_entry()，而要直接调用 create_proc_entry()。特殊文件/proc/kcore 代表着映射到系统空间的物理内存，其起点为 PAGE_OFFSET，即 0x0000 0000，而 high_memory 则为系统的物理内存在系统空间映射的终点。

创建了这些直接在/proc 目录中的特殊文件以后，proc_root_init()还要在/proc 目录中创建一些子目录，如“ net”、“ fs”、“ driver”、“ bus” 等等。这些子目录都是通过 proc_mkdir()创建的。其代码在 fs/proc/generic.c 中，不过它与 create_proc_entry()在参数 mode 中的 S_IFDIR 为 1 而 S_IALLUGO 为 0 时相同，所以我们不在这里列出它的代码了。

还有一个很特殊的子目录“ self “，/proc/self 代表着这个节点受到访问时的当前进程。也就是说，谁访问这个节点，它就代表谁，它总是代表着访问这个节点的进程自己。系统中的每一个进程在/proc 目录中都有一个以其进程号为节点名的子目录，在子目录中则又有 cmdline、cpu、cwd、environ 等节点，反映着该进程各方面的状态和信息。其中多数节点是特殊文件，有的却是目录节点。如 cwd 就是个目录节点，连接至该进程的“当前工作目录”；而 cmdline 则为启动该进程的可执行程序时的命令行。这样，特权用户可以在运行中打开任何一个进程的有关文件节点或目录节点读取该进程各方面的信息。一般用户也可以用自己的 pid 组装起路径名来获取有关其自身的信息，或者就通过/proc/self 来获取有关其自身的信息。读者不妨在机器上试一下“ more /proc/self/cmdline” 命令行，看看是什么结果。这个子目录的特殊之处还在于：它并没有一个固定的 proc_dir_entry 数据结构，也没有固定的 inode 结构，而是在需要时临时予以生成。后面我们还会回到这个话题。

上述这些子目录基本上（除 self 以外）都是最底层的目录节点，在它们下面就只有文件而再没有其它目录节点了。但是/proc/tty 却是一棵子树，在这个节点下面还有其它目录节点，所以专门有个函数 proc_tty_init()用来创建这棵子树，其代码在 fs/proc/proc_tty.c 中：

==================== fs/proc/proc_tty.c 169 181 ====================

[proc_root_init()>proc_tty_init()]

```
/*
 * Called by proc_root_init() to initialize the /proc/tty subtree
 */
void __init proc_tty_init(void)
{
	if (!proc_mkdir("tty", 0))
		return;
	proc_tty_ldisc = proc_mkdir("tty/ldisc", 0);
	proc_tty_driver = proc_mkdir("tty/driver", 0);

	create_proc_read_entry("tty/ldiscs", 0, 0, tty_ldiscs_read_proc,NULL);
	create_proc_read_entry("tty/drivers", 0, 0, tty_drivers_read_proc,NULL);
}
```

此外，如果系统不采用传统的基于主设备号 / 次设备号的/dev 设备（文件）目录，而采用树状的设备目录/device_tree，则还要在 proc_root_init()中创建起/device_tree 子树。这是由 proc_device_tree_init()完成的，其代码在 fs/proc/proc_device.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/proc/proc_devtree.c 122 139 ====================

[proc_root_init()>proc_device_tree_init()]

```
/*
 * Called on initialization to set up the /proc/device-tree subtree
 */
void proc_device_tree_init(void)
{
     struct device_node *root;
     if ( !have_of )
          return;
     proc_device_tree = proc_mkdir("device-tree", 0);
     if (proc_device_tree == 0)
          return;
     root = find_path_device("/");
     if (root == 0) {
          printk(KERN_ERR "/proc/device-tree: can't find root\n");
          return;
     }
     add_node(root, proc_device_tree);
}
```

我们在这里并不关心设备驱动，所以不深入去看 find_device_tree()和 add_node()的代码，不过读者从这两个函数的名称和调用参数可以猜到它们的作用。

如前所述，系统中的每个进程在/proc 目录中都有个以其进程号为节点名的子目录，但是这些子目录并不是在系统初始化的阶段创建的，而是要到/proc 节点受到访问时临时地生成出来。只要想想进程的创建／消失是多么的频繁，这就不足为怪了。

除这些由内核本身创建并安装的节点以外，“可安装模块”也可以根据需要通过 proc_register()在/proc 目录中创建其自己的节点，从而在模块与进程之间架起桥梁。可安装模块可以通过两种途径架设起与进程之间的桥梁，其一是通过在/dev 目录中创建一个设备文件节点，其二就是在/proc 目录中创建若干特殊文件。在老一些的版本中，可安装模块通过一个叫 proc_register_dynamic()的函数来创建 proc 文件，但是实际上可安装模块并不比进程更为动态，所以现在已经（通过宏定义）统一到了 proc_register()。当可安装模块需要在/proc 目录中创建一个特殊文件时，先准备好它自己的 inode_operations 结构和 file_operations 结构，再准备一个 proc_dir_entry 结构，然后调用 proc 将其“登记”到/proc 目录中。这就是设计和实现设备驱动程序的两种途径之一。所以，proc_register()对于要开发设备驱动程序的读者来说是一个非常重要的函数。我们在有关设备驱动的章节中还会回到这个话题。

这里要着重指出，通过 proc_register()以及 proc_mkdir()登记的是一个 proc_dir_entry 结构。结构中包含了 dentry 结构和 inode 结构所需的大部分信息，但是它既不是 dentry 结构也不是 inode 结构。同时，代表着/proc 的数据结构 proc_root 也是一个 proc_dir_entry 结构，所以由此而形成的是一棵以 proc_root 为根的树，树中的每个节点都是一个 proc_dir_entry 结构。

可是，对于文件系统的操作，如 path_walk()等等，所涉及的却是 dentry 结构和 inode 结构，这两个方面是怎样统一起来的呢？我们在前面看到，proc 文件系统的根节点/proc 有 inode 结构，这是在 proc_read_super()中通过 proc_get_inode()分配并且根据 proc_root 的内容而设置的。同时，这个节点也有 dentry 结构，这是在 proc_read_super()中通过 d_alloc_root()创建的，并且 proc 文件系统的 super_block

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构中的指针 s_root 就指向这个 dentry 结构(这个 d_entry 结构中的指针 d_inode 则指向其 inode 结构)。所以，proc 文件系统的根节点是一个“正常”的节点。在 path_walk 中首先会到达这个节点，从这以后，就由这个节点在其 inode_operations、file_operations 以及 dentry_operations 数据结构中提供的有关函数接管了进一步的操作。后面读者会看到，这些函数会临时为/proc 子树中其它的节点生成出 inode 结构来。当然，其依据就是该节点的 proc_dir_entry 结构。

下面，我们通过几个具体的情景来看 proc 文件系统中的文件操作。

第一个情景是对/proc/loadavg 的访问，这个文件提供有关系统在过去 1 分钟、5 分钟和 15 分钟内的平均负荷的统计信息。这个文件只支持读操作，其 proc_dir_entry 结构是在 proc_misc_init()中通过 create_proc_read_entry()创建的。首先，当然是通过系统调用 open()打开这个文件，为此我们要重温一下 path_walk()中的有关段落。在这个函数中，当沿着路径名搜索到了一个中间节点时，数据结构 nameidata 中的指针 dentry 指向这个中间节点的 dentry 结构，并企图继续向前搜索，而下一个节点名则在一个 qstr 数据结构 this 中。就我们这个情景而言，下一个节点已经是路径名中的最后一个节点，所以转到了 last_component 标号处。在确认了要访问的正是这个节点本身（而不是其父节点），并且节点名并非“.”或“..”以后，就先通过 cached_lookup()在内存中寻找该节点的 dentry 结构，如果这个结构尚未创建则进而通过 real_lookup()在文件系统中从其父节点开始寻找并为之创建起 dentry（以及 inode）结构。见 fs/namei.c 中的以下几行代码：

==================== fs/namei.c 558 566 ====================

```
		dentry = cached_lookup(nd->dentry, &this, 0);
		if (!dentry) {
			dentry = real_lookup(nd->dentry, &this, 0);
			err = PTR_ERR(dentry);
			if (IS_ERR(dentry))
				break;
		}
		while (d_mountpoint(dentry) && __follow_down(&nd->mnt, &dentry))
			;
```

在我们这个情景里，path_walk()首先发现/proc 节点是个安装节点，而从所安装的 super_block 结构中取得了 proc 文件系统根节点的 dentry 结构。如前所述，从这个意义上说这个节点是正常的文件系统根节点。所以，nd->dentry 就指向该节点的 dentry 结构，而 this 中则含有下一个节点名“loadavg”。然后，先通过 cached_lookup()看看下一个节点的 dentry 结构是否已经建立在内存中，如果没有就要通过 real_lookup()从设备上读入该节点的目录项（以及索引节点）并在内存中为之创建起它的 dentry 结构。但是，那只是就常规的文件系统而言，而现在的节点/proc 已经落在特殊的 proc 文件系统内，情况就不同了，先重温一下 real_lookup()中的有关代码：

==================== fs/namei.c 268 269 ====================

```
static struct dentry * real_lookup(struct dentry * parent, struct qstr * name, int flags)
{
```

==================== fs/namei.c 281 282 ====================

```
	result = d_lookup(parent, name);
	if (!result) {
```

==================== fs/namei.c 310 310 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

可见，在内存中不能发现目标节点的 dentry 结构时，到底怎么办取决于其父节点的 inode 结构中的指针 i_op 指向哪一个 inode_operations 数据结构以及这个结构中的函数指针 lookup。对于节点/proc，它的 i_op 指针指向 proc_root_inode_operations，这是在它的 proc_dir_entry 结构 proc_root 中定义好了的，具体定义见文件 fs/proc/root.c：

==================== fs/proc/root.c 79 94 ====================

```
/*
 * The root /proc directory is special, as it has the
 * <pid> directories. Thus we don't use the generic
 * directory handling functions for that..
 */
static struct file_operations proc_root_operations = {
	read:		generic_read_dir,
	readdir:	proc_root_readdir,
};

/*
 * proc root can do almost nothing..
 */
static struct inode_operations proc_root_inode_operations = {
	lookup:		proc_root_lookup,
};
```

我们在这里也一起列出了它的 file_operations 结构。从中可以看出，如果打开/proc 并通过系统调用 readdir()或 getdents()读取目录的内容（如命令“ ls”所做的那样），则调用的函数为 proc_root_readdir()。对于我们这个情景，则只是继续向前搜索，因而所调用的函数是 proc_root_lookup()，其代码在 fs/proc/root.c 中：

==================== fs/proc/root.c 49 62 ====================

```
static struct dentry *proc_root_lookup(struct inode * dir, struct dentry * dentry)
{
	if (dir->i_ino == PROC_ROOT_INO) { /* check for safety... */
		int nlink = proc_root.nlink;

		nlink += nr_threads;

		dir->i_nlink = nlink;
	}

	if (!proc_lookup(dir, dentry))
		return NULL;

	return proc_pid_lookup(dir, dentry);
}
```

参数 dir 指向父节点即/proc 的 inode 结构，这个 inode 结构中的 nlink 字段也是特殊的，它的数值

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

等于当前系统中进程（以及线程）的数量。由于这个数量随时都可能在变，所以每次调用 proc_root_lookup()时都要根据当时的情景予以更新。这里 nr_threads 是内核中的一个全局量，反映着系统中的进程数量。另一方面，由于系统中的进程数量不会降到 0，这个字段的数值也不可能为 0。

/proc 目录中的节点可以分为两类。一类是节点的 proc_dir_entry 结构已经向 proc_root() “登记”，而挂入了其队列中；另一类则对应于当前系统中的各个进程而并不存在 proc_dir_entry 结构。前者需要通过 proc_lookup()找到其 proc_dir_entry 结构而设置其 dentry 结构并创建其 inode 结构。后者则需要根据节点名（进程号）在系统中找到进程的 task_struct 结构，再设置节点的 dentry 结构并创建 inode 结构。显然，/proc/loadavg 属于前者，所以我们继续看 proc_lookup()的代码，它在文件 fs/proc/generic.c 中：

==================== fs/proc/generic.c 237 271 ====================

[proc_root_lookup()>proc_lookup()]

```
/*
 * Don't create negative dentries here, return -ENOENT by hand
 * instead.
 */
struct dentry *proc_lookup(struct inode * dir, struct dentry *dentry)
{
     struct inode *inode;
     struct proc_dir_entry * de;
     int error;

     error = -ENOENT;
     inode = NULL;
     de = (struct proc_dir_entry *) dir->u.generic_ip;
     if (de) {
          for (de = de->subdir; de ; de = de->next) {
               if (!de || !de->low_ino)
                    continue;
               if (de->namelen != dentry->d_name.len)
                    continue;
               if (!memcmp(dentry->d_name.name, de->name, de->namelen)) {
                    int ino = de->low_ino;
                    error = -EINVAL;
                    inode = proc_get_inode(dir->i_sb, ino, de);
                    break;
               }
          }
     }

     if (inode) {
          dentry->d_op = &proc_dentry_operations;
          d_add(dentry, inode);
          return NULL;
     }
     return ERR_PTR(error);
}
```

这里的参数 dir 指向父节点即/proc 的 inode 结构，而 dentry 则指向已经分配用于目标节点的 dentry 结构。函数本身的逻辑是很简单的，proc_get_inode()的代码也已在前面看到过。至于 d_add()，则只是将 dentry 结构挂入杂凑表队列，并使 dentry 结构与 inode 结构互相挂上钩，读者在本章开头几节中也已看到过。

从 proc_lookup()一路正常返回到 path_walk()中时，沿着路径名的搜索就向前推进了一步。在我们这个情景中，路径名至此已经结束，所以 path_walk()已经完成了操作，找到了目标节点/proc/loadavg 的 dentry 结构，此后就与常规的 open()操作没有什么两样了。

打开了文件以后，就是通过系统调用 read()从文件中读。由于目标文件的 dentry 结构和 inode 结构均已建立，所以开始时的操作与常规文件的并无不同，直到根据 file 结构中的指针 f_op 找到相应的 file_operations 结构并进而找到其函数指针 read。对于 proc 文件系统，file 结构中的 f_op 指针来自目标文件 inode 结构，而 inode 结构中的这个指针又来源于目标节点的 proc_dir_entry 结构（见 proc_get_inode()的代码）。在 proc_register()的代码中可以看出，目录节点的 proc_fops 都指向 proc_dir_operations；而"普通"文件节点（如/proc/loadavg）的 proc_fops 则都指向 proc_file_operations。所以，/proc/loadavg 的 file_operations 结构为 proc_file_operations，这是在 fs/proc/generic.c 中定义的：

==================== fs/proc/generic.c 36 40 ====================

```
static struct file_operations proc_file_operations = {
	llseek:		proc_file_lseek,
	read:		proc_file_read,
	write:		proc_file_write,
};
```

可见，为读文件操作提供的函数是 proc_file_read()。这是一个为 proc 特殊文件通用的函数，其代码也在 generic.c 中：

==================== fs/proc/generic.c 46 121 ====================

```
/* buffer size is one page but our output routines use some slack for overruns */
#define PROC_BLOCK_SIZE(PAGE_SIZE - 1024)

static ssize_t
proc_file_read(struct file * file, char * buf, size_t nbytes, loff_t *ppos)
{
	struct inode * inode = file->f_dentry->d_inode;
	char 	*page;
	ssize_t	retval=0;
	int	eof=0;
	ssize_t	n, count;
	char *start;
	struct proc_dir_entry * dp;

	dp = (struct proc_dir_entry *) inode->u.generic_ip;
	if (!(page = (char*) __get_free_page(GFP_KERNEL)))
		return -ENOMEM;

	while ((nbytes > 0) && !eof)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    {
        count = MIN(PROC_BLOCK_SIZE, nbytes);

        start = NULL;
        if (dp->get_info) {
            /*
             * Handle backwards compatibility with the old net
             * routines.
             */
            n = dp->get_info(page, &start, *ppos, count);
            if (n < count)
                eof = 1;
        } else if (dp->read_proc) {
            n = dp->read_proc(page, &start, *ppos,
                          count, &eof, dp->data);
        } else
            break;

        if (!start) {
            /*
             * For proc files that are less than 4k
             */
            start = page + *ppos;
            n -= *ppos;
            if (n <= 0)
                break;
            if (n > count)
                n = count;
        }
        if (n == 0)
            break;   /* End of file */
        if (n < 0) {
            if (retval == 0)
                retval = n;
            break;
        }

        /* This is a hack to allow mangling of file pos independent
         * of actual bytes read.  Simply place the data at page,
         * return the bytes, and set `start' to the desired offset
         * as an unsigned int. - Paul.Russell@rustcorp.com.au
         */
        n -= copy_to_user(buf, start < page ? page : start, n);
        if (n == 0) {
            if (retval == 0)
                retval = -EFAULT;
            break;
        }

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            *ppos += start < page ? (long)start : n; /* Move down the file */
            nbytes -= n;
            buf += n;
            retval += n;
        }
        free_page((unsigned long) page);
        return retval;
}
```

从总体上说，这个函数的代码并没有什么特殊，对本书的读者不应成为问题。但是从中可以看出，具体的读操作是通过由节点的 proc_dir_entry 结构中的函数指针 get_info 或 read_proc 提供的。其中 get_info 是为了与老一些的版本兼容而保留的，现在已改用 read_proc。与常规的文件系统如 Ext2 相比，proc 文件系统有个特殊之处：那就是它的每个具体的文件或节点都有其自己的文件操作函数，而不像 Ext2 那样由其 file_operation 结构中提供的函数可以用于同一文件系统的所有文件。当然，这是由于在 proc 文件系统中每个节点都有其特殊性。正因为这样，proc 文件系统的 file_operations 结构中只为读操作提供一个通用的函数 proc_file_read()，而由它再进一步找到并调用具体节点所提供的 read_proc 函数。在前面 proc_misc_init()以及 create_proc_read_entry()的代码中，我们看到节点/proc/loadavg 的这个指针指向 loadavg_read_proc()，其代码在 fs/proc/proc_misc.c 中：

==================== fs/proc/proc_misc.c 86 101 ====================

```
static int loadavg_read_proc(char *page, char **start, off_t off,
                    int count, int *eof, void *data)
{
        int a, b, c;
        int len;

        a = avenrun[0] + (FIXED_1/200);
        b = avenrun[1] + (FIXED_1/200);
        c = avenrun[2] + (FIXED_1/200);
        len = sprintf(page,"%d.%02d %d.%02d %d.%02d %d/%d %d\n",
            LOAD_INT(a), LOAD_FRAC(a),
            LOAD_INT(b), LOAD_FRAC(b),
            LOAD_INT(c), LOAD_FRAC(c),
            nr_running, nr_threads, last_pid);
        return proc_calc_metrics(page, start, off, count, eof, len);
}
```

==================== fs/proc/proc_misc.c 75 84 ====================

```
static int proc_calc_metrics(char *page, char **start, off_t off,
                    int count, int *eof, int len)
{
        if (len <= off+count) *eof = 1;
        *start = page + off;
        len -= off;
        if (len>count) len = count;
        if (len<0) len = 0;
        return len;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

它的作用就是将数组 avenrun[]中积累的在过去 1 分钟、5 分钟以及 15 分钟内的系统平均 CPU 负荷等统计信息通过 sprintf()“打印”到缓冲区页面中。这些平均负荷的数值是每隔 5 秒钟在时钟中断服务程序中进行计算的。统计信息中还包括系统当前处于可运行状态（在运行队列中）的进程个数 nr_running 以及系统中进程的总数 nr_threads，还有系统中已分配使用的最大进程号 last_pid。

我们要看的第二个情景是对/proc/self/cwd 的访问。前面讲过，/proc/self 节点在受到访问时，会根据当前进程的进程号连接到/proc 目录中以此进程号为节点名的目录节点，而这个目录节点下面的 cwd 则又连接到该进程的“当前工作目录”。所以，在这短短的路径名中就有两个连接节点，而且/proc/self/cwd 是从 proc 文件系统中的节点到常规文件系统（如 Ext2）中的节点的连接。我们对目标节点即“当前工作目录”中的内容本身并不感兴趣，而只是对 path_walk()怎样两次跨越文件系统进行路径搜索感兴趣。

第一次跨越文件系统是当 path_walk 发现/proc 是个安装节点而通过__follow_down()找到所安装的 super_block 结构的过程。这方面并没有什么特殊，读者也已经熟悉了。找到了 proc 文件系统的根节点的 dentry 结构以后，nameidata 结构中的指针 dentry 指向这个数据结构，并企图继续向前搜索路径名中的下一个节点 self。由于这个节点并不是路径名中的最后一个节点，所执行的代码是从文件 fs/namei.c 中 path_walk()内的 494 行开始的：

==================== fs/namei.c 494 501 ====================

```
494          /* This does the actual lookups.. */
495          dentry = cached_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
496          if (!dentry) {
497               dentry = real_lookup(nd->dentry, &this, LOOKUP_CONTINUE);
498               err = PTR_ERR(dentry);
499               if (IS_ERR(dentry))
500                    break;
501          }
```

就所执行的代码本身而言，是与前一个情景一样的，所以最终也要通过 proc_root_lookup()调用 proc_lookup()，试图为节点建立起其 dentry 结构和 inode 结构。可是，如前所述，由于/proc/self 并没有一个固定的 proc_dir_entry 结构，所以对 proc_lookup()的调用必然会失败（返回非 0），因而会进一步调用 proc_pid_lookup()。这个函数的代码在 fs/proc/base.c 中，我们先看它的前一部分：

==================== fs/proc/base.c 907 932 ====================

[proc_root_lookup()>proc_pid_lookup()]

```
struct dentry *proc_pid_lookup(struct inode *dir, struct dentry * dentry)
{
     unsigned int pid, c;
     struct task_struct *task;
     const char *name;
     struct inode *inode;
     int len;

     pid = 0;
     name = dentry->d_name.name;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
917        len = dentry->d_name.len;
918        if (len == 4 && !memcmp(name, "self", 4)) {
919            inode = new_inode(dir->i_sb);
920            if (!inode)
921                return ERR_PTR(-ENOMEM);
922            inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;
923            inode->i_ino = fake_ino(0, PROC_PID_INO);
924            inode->u.proc_i.file = NULL;
925            inode->u.proc_i.task = NULL;
926            inode->i_mode = S_IFLNK|S_IRWXUGO;
927            inode->i_uid = inode->i_gid = 0;
928            inode->i_size = 64;
929            inode->i_op = &proc_self_inode_operations;
930            d_add(dentry, inode);
931            return NULL;
932        }
```

可见，当要找寻的节点名为“self”时内核为之分配一个空白的 inode 数据结构，并使其 inode_operations 结构指针 i_op 指向专用于/proc/self 的 proc_self_inode_operations，这个结构的定义在 fs/proc/root/c 中：

==================== fs/proc/base.c 902 905 ====================

```
902  static struct inode_operations proc_self_inode_operations = {
903        readlink:proc_self_readlink,
904        follow_link:  proc_self_follow_link,
905  };
```

为此类节点建立的 inode 结构有着特殊的节点号，这是由进程的 pid 左移 16 位以后与常数 PROC_PID_INO 相或而形成的，常数 PROC_PID_INO 则定义为 2。

从 proc_root_lookup()返回到 path_walk()中以后，接着要检查和处理两件事，第一件是新找到的节点是否为安装点；第二件就是它是否是一个连接节点。这正是我们在这里所关心的，因为/proc/self 就是个连接节点。继续看 path_walk()中的下面两行（fs/namei.c）：

==================== fs/namei.c 514 515 ====================

```
514            if (inode->i_op->follow_link) {
515                err = do_follow_link(dentry, nd);
```

对于连接节点，通过其 inode 结构和 inode_operations 结构提供的函数指针 follow_link 为非 0。就/proc/self 而言，由于 proc_pid_lookup()的执行，其函数指针 follow_link 已经指向了 proc_self_follow_link()，其代码在文件 fs/proc/base.c 中：

==================== fs/proc/base.c 895 900 ====================

```
895  static int proc_self_follow_link(struct dentry *dentry, struct nameidata *nd)
896  {
897        char tmp[30];
898        sprintf(tmp, "%d", current->pid);
899        return vfs_follow_link(nd,tmp);
900  }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

它通过 vfs_follow_link()寻找以当前进程的 pid 的字符串为相对路径名的节点，找到后就使 nameidata 结构中的指针 dentry 指向它的 dentry 结构。读者已经看到过 vfs_follow_link()的代码，这里就不重复了。只是要指出，在 vfs_follow_link()中将会递归地调用 path_walk()来寻找连接的目标节点，所以又会调用其父节点/proc 的 lookup 函数，即 proc_root_lookup()，不同的只是这次寻找的不是“self”，而是当前进程的 pid 字符串。这一次，在 proc_root_lookup()中对 proc_lookup()的调用同样会因为在 proc_root 的 subdir 队列中找不到相应的 proc_dir_entry 结构而失败，所以也要进一步调用 proc_pid_lookup()寻找（见上面 proc_root_lookup()的代码）。可是，这一次的节点名就不是“self”了，刚调用的 proc_self_follow_link()已经将当前进程的进程号转化为字符串形式，所以在 proc_pid_lookup()中所走的路线也不同了，我们看这个函数的后半部：

==================== fs/proc/base.c 933 971 ====================

[proc_root_lookup()>proc_pid_lookup()]

```
	while (len-- > 0) {
		c = *name - '0';
		name++;
		if (c > 9)
			goto out;
		if (pid >= MAX_MULBY10)
			goto out;
		pid *= 10;
		pid += c;
		if (!pid)
			goto out;
	}

	read_lock(&tasklist_lock);
	task = find_task_by_pid(pid);
	if (task)
		get_task_struct(task);
	read_unlock(&tasklist_lock);
	if (!task)
		goto out;

	inode = proc_pid_make_inode(dir->i_sb, task, PROC_PID_INO);

	free_task_struct(task);

	if (!inode)
		goto out;
	inode->i_mode = S_IFDIR|S_IRUGO|S_IXUGO;
	inode->i_op = &proc_base_inode_operations;
	inode->i_fop = &proc_base_operations;
	inode->i_nlink = 3;
	inode->i_flags|=S_IMMUTABLE;

	dentry->d_op = &pid_base_dentry_operations;
	d_add(dentry, inode);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        return NULL;
out:
        return ERR_PTR(-ENOENT);
}
```

这个函数将节点名转换成一个无符号整数，然后以此为 pid 从系统中寻找是否存在相应的进程。如果找到了相应的进程，就通过 proc_pid_make_inode()为之创建一个 inode 结构，并初始化已经分配的 dentry 结构。这个函数的代码在文件 fs/proc/base.c 中，我们就不看了。同时，还要使 inode 结构中的 inode_operations 结构指针 i_op 指向 proc_base_inode_operations，而 file_operations 结构指针 i_fop 则指向 proc_base_operations。此外，相应 dentry 结构中的指针 d_op 则指向 pid_base_dentry_operations。从这里也可以看出，在 proc 文件系统中几乎每个节点都有其自己的 file_operations 结构和 inode_operations 结构。

于是，从 proc_follow_link()返回时，nd->dentry 已指向代表着当前进程的目录节点的 dentry 结构。这样，当 path_walk()开始新一轮的循环时，就从这个节点（而不是/proc/self）继续向前搜索了。下一个节点是“ cwd”，这一次所搜索的节点已经是路径名中的最后一个节点，所以如同第一个情景中那样转到了标号为 last_component 的地方。但是同样也是在 real_lookup()中通过其父节点的 inode_operations 结构中的 lookup 函数指针执行实际的操作，而现在这个数据结构就是 proc_base_inode_operaions，定义于 fs/proc/base.c：

==================== fs/proc/base.c 881 883 ====================

```
static struct inode_operations proc_base_inode_operations = {
        lookup:         proc_base_lookup,
};
```

函数 proc_base_lookup()的代码在同一文件 fs/proc/base.c

==================== fs/proc/base.c 783 874 ====================

```
static struct dentry *proc_base_lookup(struct inode *dir, struct dentry *dentry)
{
        struct inode *inode;
        int error;
        struct task_struct *task = dir->u.proc_i.task;
        struct pid_entry *p;

        error = -ENOENT;
        inode = NULL;

        for (p = base_stuff; p->name; p++) {
                if (p->len != dentry->d_name.len)
                        continue;
                if (!memcmp(dentry->d_name.name, p->name, p->len))
                        break;
        }
        if (!p->name)
                goto out;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        error = -EINVAL;
        inode = proc_pid_make_inode(dir->i_sb, task, p->type);
        if (!inode)
                goto out;

        inode->i_mode = p->mode;
        /*
         * Yes, it does not scale. And it should not. Don't add
         * new entries into /proc/<pid>/ without very good reasons.
         */
        switch(p->type) {
                case PROC_PID_FD:
                        inode->i_nlink = 2;
                        inode->i_op = &proc_fd_inode_operations;
                        inode->i_fop = &proc_fd_operations;
                        break;
                case PROC_PID_EXE:
                        inode->i_op = &proc_pid_link_inode_operations;
                        inode->u.proc_i.op.proc_get_link = proc_exe_link;
                        break;
                case PROC_PID_CWD:
                        inode->i_op = &proc_pid_link_inode_operations;
                        inode->u.proc_i.op.proc_get_link = proc_cwd_link;
                        break;
                case PROC_PID_ROOT:
                        inode->i_op = &proc_pid_link_inode_operations;
                        inode->u.proc_i.op.proc_get_link = proc_root_link;
                        break;
                case PROC_PID_ENVIRON:
                        inode->i_fop = &proc_info_file_operations;
                        inode->u.proc_i.op.proc_read = proc_pid_environ;
                        break;
                case PROC_PID_STATUS:
                        inode->i_fop = &proc_info_file_operations;
                        inode->u.proc_i.op.proc_read = proc_pid_status;
                        break;
                case PROC_PID_STAT:
                        inode->i_fop = &proc_info_file_operations;
                        inode->u.proc_i.op.proc_read = proc_pid_stat;
                        break;
                case PROC_PID_CMDLINE:
                        inode->i_fop = &proc_info_file_operations;
                        inode->u.proc_i.op.proc_read = proc_pid_cmdline;
                        break;
                case PROC_PID_STATM:
                        inode->i_fop = &proc_info_file_operations;
                        inode->u.proc_i.op.proc_read = proc_pid_statm;
                        break;
                case PROC_PID_MAPS:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
			inode->i_fop = &proc_maps_operations;
			break;
#ifdef CONFIG_SMP
		case PROC_PID_CPU:
			inode->i_fop = &proc_info_file_operations;
			inode->u.proc_i.op.proc_read = proc_pid_cpu;
			break;
#endif
		case PROC_PID_MEM:
			inode->i_op = &proc_mem_inode_operations;
			inode->i_fop = &proc_mem_operations;
			break;
		default:
			printk("procfs: impossible type (%d)",p->type);
			iput(inode);
			return ERR_PTR(-EINVAL);
	}
	dentry->d_op = &pid_dentry_operations;
	d_add(dentry, inode);
	return NULL;

out:
	return ERR_PTR(error);
}
```

这里用到一个全局性的数组 base_stuff[]，有关的定义在 fs/proc/base.c 中给出：

==================== fs/proc/base.c 477 519 ====================

```
477  struct pid_entry {
478  	int type;
479  	int len;
480  	char *name;
481  	mode_t mode;
482  };
483
484  enum pid_directory_inos {
485  	PROC_PID_INO = 2,
486  	PROC_PID_STATUS,
487  	PROC_PID_MEM,
488  	PROC_PID_CWD,
489  	PROC_PID_ROOT,
490  	PROC_PID_EXE,
491  	PROC_PID_FD,
492  	PROC_PID_ENVIRON,
493  	PROC_PID_CMDLINE,
494  	PROC_PID_STAT,
495  	PROC_PID_STATM,
496  	PROC_PID_MAPS,
497  	PROC_PID_CPU,
```

```
        PROC_PID_FD_DIR = 0x8000,   /* 0x8000-0xffff */
};

#define E(type,name,mode) {(type),sizeof(name)-1,(name),(mode)}
static struct pid_entry base_stuff[] = {
  E(PROC_PID_FD,    "fd",          S_IFDIR|S_IRUSR|S_IXUSR),
  E(PROC_PID_ENVIRON,    "environ",     S_IFREG|S_IRUSR),
  E(PROC_PID_STATUS,     "status", S_IFREG|S_IRUGO),
  E(PROC_PID_CMDLINE,    "cmdline",     S_IFREG|S_IRUGO),
  E(PROC_PID_STAT, "stat",         S_IFREG|S_IRUGO),
  E(PROC_PID_STATM,      "statm",  S_IFREG|S_IRUGO),
#ifdef CONFIG_SMP
  E(PROC_PID_CPU,   "cpu",         S_IFREG|S_IRUGO),
#endif
  E(PROC_PID_MAPS, "maps",         S_IFREG|S_IRUGO),
  E(PROC_PID_MEM,   "mem",         S_IFREG|S_IRUSR|S_IWUSR),
  E(PROC_PID_CWD,   "cwd",         S_IFLNK|S_IRWXUGO),
  E(PROC_PID_ROOT, "root",         S_IFLNK|S_IRWXUGO),
  E(PROC_PID_EXE,   "exe",         S_IFLNK|S_IRWXUGO),
  {0,0,NULL,0}
};
#undef E
```

这样，在 proc_base_lookup()中只要在这个数组中逐项比对，就可以找到"cwd"所对应的类型，即相应 inode 号中的低 16 位，以及"文件"的模式。然后，在基于这个类型的 switch 语句中，对于所创建的 inode 结构进行具体的设置。对于代表着进程的某方面属性或状态的这些 inode 结构，其 union 部分被用作一个 proc_inode_info 结构 proc_i，其定义见于 include/linux/proc_fs_i.h：

==================== include/linux/proc_fs_i.h 1 9 ====================

```
struct proc_inode_info {
        struct task_struct *task;
        int type;
        union {
             int (*proc_get_link)(struct inode *, struct dentry **, struct vfsmount **);
             int (*proc_read)(struct task_struct *task, char *page);
        } op;
        struct file *file;
};
```

结构中的指针 task 在 proc_pid_make_inode()中设置成指向 inode 结构所代表进程的 task_struct 结构。

对于节点"cwd"，要特别加以设置的内容有两项。第一是将 inode 结构中的指针 i_op 设置成指向 proc_pid_link_inode_operations 数据结构；第二是将上述 proc_inode_info 结构中的函数指针 proc_get_link 指向 proc_cwd_link()。此外，就没有什么特殊之处了。数据结构 proc_pid_link_inode_operaions 定义于 fs/proc/base.c 中：

==================== fs/proc/base.c 472 475 ====================

```
static struct inode_operations proc_pid_link_inode_operations = {
        readlink: proc_pid_readlink,
```

```
        follow_link:  proc_pid_follow_link
};
```

从 proc_base_lookup()经由 real_lookup()返回到 path_walk()时，nameidata 结构中的指针 dentry 已经指向了这个特定“ cwd”节点的 dentry 结构。但是接着同样要受到对其 inode 结构中的 i_op 指针以及相应 inode_operations 结构中的指针 follow_link 的检验，看 path_walk()中的相关代码（fs/namei.c）：

==================== fs/namei.c 567 570 ====================

```
            inode = dentry->d_inode;
            if ((lookup_flags & LOOKUP_FOLLOW)
               && inode && inode->i_op && inode->i_op->follow_link) {
                err = do_follow_link(dentry, nd);
```

读者刚才已经看到，这个 inode 结构的指针 follow_link 非 0，并且指向 proc_cwd_link()，其代码在 fs/proc/base.c 中：

==================== fs/proc/base.c 85 103 ====================

```
85   static int proc_cwd_link(struct inode *inode, struct dentry **dentry, struct vfsmount **mnt)
86   {
87        struct fs_struct *fs;
88        int result = -ENOENT;
89        task_lock(inode->u.proc_i.task);
90        fs = inode->u.proc_i.task->fs;
91        if(fs)
92             atomic_inc(&fs->count);
93        task_unlock(inode->u.proc_i.task);
94        if (fs) {
95             read_lock(&fs->lock);
96             *mnt = mntget(fs->pwdmnt);
97             *dentry = dget(fs->pwd);
98             read_unlock(&fs->lock);
99             result = 0;
100            put_fs_struct(fs);
101       }
102       return result;
103  }
```

如前所述，节点的 inode 中的 union 用作一个 proc_inode_info 结构，其中的指针 task 指向相应进程的 task_struct 结构，进而可以得到这个进程的 fs_struct 结构，而这个数据结构中的指针 pwd 即指向该进程的“当前工作目录”的 dentry 结构，同时指针 pwdmnt 指向该目录所在设备安装时的 vfsmount 结构。注意，这里的参数 dentry 和 mnt 都是双重指针，所以第 96 行和第 97 行实际上改变了 nameidata 结构中的 dentry 和 mnt 两个指针。这样，当从 proc_cwdlink()经由 do_follow_link()返回到 path_walk()中时，nameidata 结构中的指针已经指向最终的目标，即当前进程的当前工作目录。从这以后的操作就与常规的文件系统完全一样了。从这个情景可以看出，对于 proc 文件系统中的一些路径，其有关的数据结构以及这些数据结构之间的连接是非常动态的，每次都要在 path_walk()的过程中逐层地临时建立，而不像在常规文件系统如 Ext2 中那样相对静态。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

通过这两个情景，读者应该已经对 proc 文件系统的文件操作有了基本的了解和理解，自己不妨再读几段代码以加深理解，我们建议读者读一下对/proc/meminfo 和/proc/self/maps 的访问，因为这不仅可以加深对 proc 文件系统的理解，还可以帮助巩固对存储管理的理解。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第6章 传统的 Unix 进程间通信

6.1 概述

对于多用户、多进程的操作系统来说，进程间通信（IPC）是一项非常重要、甚至必不可少的基本手段和设施。在一个多进程操作系统所提供的运行环境下，可以通过两种不同的途径，或者说采用两种不同的策略，来建立起复杂的大型应用系统。一种途径是通过一个孤立的、大型的、复杂的进程提供所需的全部功能，另一种途径则是通过由若干互相联系的、小型的、相对简单的进程所构成的组合来提供所需的功能。早期的操作系统往往倾向于前者，而 Unix 及其衍生的各种系统则倾向于后者。这种基本方法和策略的改变正是 Unix 操作系统在程序设计领域中引起革命性转变的结果。相比之下，后者这种方法具有很大的好处：

- 首先，这种途径使应用软件更加模块化，每个进程所执行的程序可以分别地设计、实现、调试和维护。
- 其次，由于每个进程都有其独立的地址空间，而相互间的通信则通过明确定义的进程间通信手段和界面来完成，因而使得各个进程都得到保护，在相当程度上排除了互相干扰的可能性，从而增加了系统的可靠性和稳定性。
- 而且，这种途径还改善了系统规模的可扩充性。例如，在多处理器系统中，这些进程可以在不同的处理器上运行。推而广之，这些进程还可以在多台计算机上运行，并且这些计算机并不非得是在同一地域，从而形成“分布式处理”的概念。
- 最后，就像用 7 个音符可以组合出无数动听的旋律一样，用若干可执行程序也可以灵活地搭建出很复杂、功能很强的新应用。从这个角度来看，这种途径既促进了软件的模块化，也提高了软件的“复用性”。

当然，取得这些好处也是有代价的，这种途径也有缺点。首先，从全局来看，这种途径要占用更多的资源，并且增加了 CPU 运行时的系统开销，使得总体上的运行效率可能会有所下降。其次，由于每个进程都独立地接受调度，使得进程运行的时序在某些情况下成为问题，需要通过一些特殊的进程间通信手段才能保持同步。最后，这种途径要求操作系统提供充分的进程间通信的手段和设施。但是，这些都只是前进道路上所遇到的问题。相比之下，这种途径的优点远远超过其缺点，而且随着硬件技术的进步（如内存容量、处理器速度等等），第一条缺点实际上已是微不足道了。事实上，随着应用软件的日益复杂和规模的日益庞大，通过孤立、大型、复杂的单进程途径来实现应用所需的功能往往已经不现实了。

由此可见，进程间通信在现代操作系统中起着至关重要的作用。可以这样说，没有 Unix 的进程间通信手段就不会有所谓“ Unix 环境”，即 Unix 独特的运行环境和程序设计环境。从另一个角度来说，同任何一种新技术的出现一样，一旦人们认识到上述途径的优越性及其对进程间通信手段的需求，这些手段就一定会应运而生。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

那么，Unix（从而 Linux）向应用软件提供一些什么样的进程间通信手段呢？这里也有个发展过程。早期的 Unix 提供了以下一些手段：

- 管道（Pipe）。父进程与子进程之间，或者两个兄弟进程之间，可以通过系统调用建立起一个单向的通信管道。但是，这种管道只能由父进程来建立，所以对于子进程来说是静态的，与生俱来的。管道两端的进程各自都将该管道视作一个文件。一个进程往管道中写的内容由另一个进程从管道中读取，通过管道传递的内容遵循“先入先出”（FIFO）的规则。每个管道都是单向的，需要双向通信时就要建立起两个管道。
- 信号（Signal）。读者已经在第 4 章中看到过信号的运用。严格说来，signal 这种手段并不是专为进程间通信而设置的，它也用于内核与进程之间的通信（不过内核只能向进程发送信号，而不能接收信号）。一般来说，signal 是对“中断”这种概念在软件层次上的模拟（所以亦称“软中断”），其中信号的发送者相当于中断源，而接收者则相当于处理器，所以必须是一个进程。就像在多处理器系统中一个处理器通常都能向另一个处理器发出中断请求一样，一个进程也可以向其它进程发出信号，此时信号就成了一种进程间通信的手段。
- 跟踪（Trace）。一个进程可以通过系统调用 ptrace()读／写其子进程地址空间中的内容，从而达到跟踪子进程执行的目的。

这几项进程间通信手段都只能用于父进程与子进程之间，或者两个兄弟进程之间。信号的使用虽然并未限制在父子进程之间，但发送信号时需要用到对方的 pid，而一般只有父子进程之间才知道对方的 pid，所以实际上还是只能用于父子进程之间。另一方面，对于子进程来说管道机制是静态的，跟踪则是单向的。在实际使用中，常常需要在并非这些“近亲”的进程之间动态地建立通信管道，所以后来又增设了一种新的管道：

- 命名管道（named pipe）。命名管道以 FIFO 文件的形式出现在文件系统中，所以任何进程都可以通过使用其文件名来“打开”该管道，然后进行读写。这样，管道的使用就不再局限于“近亲”之间了。从这个意义上说，命名管道是管道的推广。

在 AT&T 的 Unix 系统 V 中，主要为了更好地支持商业应用中的“事务处理”，又增加了三种进程间通信手段，常常合在一起称作“ System V IPC”：

- 报文（Message）队列。一个进程可以通过系统调用设立一个报文队列，然后任何进程都可以通过系统调用向这个队列发送“消息”或从队列接收“消息”，从而以进程间“报文传递 ”（Message Passing）的形式实现通信。
- 共享内存。一个进程可以通过系统调用设立一片共享内存区，然后其它进程就可以通过系统调用将该存储区映射到其用户地址空间中。此后，就可以像正常的内存访问一样读／写该共享区间了。共享内存是一种快速而有效的进程间通讯手段，但是并不像其它一些手段那样，可以在一旦写入后就唤醒正在睡眠中等待读取的进程，所以常常要与其它手段配合使用。
- 信号量（Semaphore）。第 4 章中讲过内核中使用的信号量机制，而系统 V 进程间通信手段中的信号量则将这种机制推广到了用户空间。

与 AT&T 对 Unix 原有进程间通信手段的扩充与增强相平行，在 BSD Unix 中也对此作了重要的扩充：

- 插口（Socket）。从语义的角度来说，Socket 与命名管道是很相似的，但其重要之处在于 Socket

不仅可以用来实现同一台计算机上的进程间通信，还可以用来实现分布于不同计算机中的进程通过网络进行的通信。这样，就提供了一种统一的、更为一般的进程间通信模式。如果说“命名管道”把“管道”这种原来只适用近亲的手段推广到了同一台计算机中的任意进程之间，则 Socket 又进一步将其推广至计算机网络中的任意进程之间。从这个意义上讲，Socket 成了最一般、最普遍适用的进程间通信手段和机制。事实上，现在有些 Unix 系统中的管道机制也反过来改成通过 Socket 来实现。

上述的这些进程间通信机制都由 Linux“兼容并蓄”继承了下来。事实上这些机制大都是在“可移植操作系统接口”标准 POSIX.1 中规定要具备的。此外，在 Unix 发展过程中也出现过一些其它的进程间通信机制，但并没有为 Linux 所采用，我们就不作介绍了。

由于篇幅的原因，我们把进程间通信分成第 6 章、第 7 章两章。本章主要介绍早期 Unix 的通信机制以及 System V IPC；下一章“基于 Socket 的进程间通信”则集中介绍插口（Socket）。

6.2 管道和系统调用 pipe()

管道机制的主体是系统调用 pipe()，但是由 pipe()所建立的管道的两端都在同一进程中，所以必须在 fork()的配合下，才能在父子进程之间或两个子进程之间建立起进程间的通信管道。

我们先来看系统调用 pipe()。

由于管道两端都是以（已打开）文件的形式出现在相关的进程中，在具体实现上也是作为匿名文件来实现的，所以 pipe()的代码与文件系统密切相关。

先看系统调用的入口 sys_pipe()，其代码在 arch/i386/kernel/sys_i386.c 中：

==================== arch/i386/kernel/sys_i386.c 25 40 ====================

```
/*
 * sys_pipe() is the normal C calling standard for creating
 * a pipe. It's not the way Unix traditionally does this, though.
 */
asmlinkage int sys_pipe(unsigned long * fildes)
{
	int fd[2];
	int error;

	error = do_pipe(fd);
	if (!error) {
		if (copy_to_user(fildes, fd, 2*sizeof(int)))
			error = -EFAULT;
	}
	return error;
}
```

这里由 do_pipe()建立起一个管道，通过作为调用参数的数组 fd[]返回代表着管道两端的两个已打开文件号，再由 copy_to_user()将数组 fd[]复制到用户空间。显然，do_pipe()是这个系统调用的主体，其代码在 fs/pipe.c 中，我们分段阅读：

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

==================== fs/pipe.c 509 541 ====================

[sys_pipe()>do_pipe()]

```
509 int do_pipe(int *fd)
510 {
511     struct qstr this;
512     char name[32];
513     struct dentry *dentry;
514     struct inode * inode;
515     struct file *f1, *f2;
516     int error;
517     int i,j;
518
519     error = -ENFILE;
520     f1 = get_empty_filp();
521     if (!f1)
522         goto no_files;
523
524     f2 = get_empty_filp();
525     if (!f2)
526         goto close_f1;
527
528     inode = get_pipe_inode();
529     if (!inode)
530         goto close_f12;
531
532     error = get_unused_fd();
533     if (error < 0)
534         goto close_f12_inode;
535     i = error;
536
537     error = get_unused_fd();
538     if (error < 0)
539         goto close_f12_inode_i;
540     j = error;
541
```

在“文件系统”一章中读者已经看到，进程对每个已打开文件的操作都是通过一个 file 数据结构进行的，只有在由同一进程按相同模式重复打开同一文件时才共享同一个数据结构。一个管道实际上就是一个无形（只存在于内存中）的文件，对这个文件的操作要通过两个已打开文件进行，分别代表该管道的两端。虽然最初创建时一个管道的两端都在同一进程中，但是在实际使用时却总是分别在两个不同的进程（通常是父、子进程）中，所以，管道的两端不能共享同一个 file 数据结构，而要为之各分配一个 file 数据结构。代码中 520 行和 524 行调用 get_empty_filp()的目的就是为管道的两端 f1 和 f2 各分配一个 file 数据结构。get_empty_filp()的代码以及 file 结构的定义可参看“文件系统”一章，这里不再重复。只是要指出，这个数据结构只是代表着一个特定进程对某个文件操作的现状，而并不代表这个文件本身的状态。例如，结构中的成分 f_pos 就表示该进程在此文件中即将进行读 / 写的起始位置，当不同的进程分别打开同一文件进行读写时，最初此位置都是 0，以后就可能各不相同了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

同时，每个文件都是由一个 inode 数据结构代表的。虽然一个管道实际上是一个无形的文件，它也得要有一个 inode 数据结构。由于这个文件在创建管道之前并不存在，所以需要在创建管道时临时创建起一个 inode 结构，这就是代码中第 528 行调用 get_pipe_inode()的目的。实际上，创建一个管道的过程主要就是创建这么一个文件的过程。函数 get_pipe_inode()的代码在 fs/pipe.c 中：

==================== fs/pipe.c 476 507 ====================

[sys_pipe()>do_pipe()>get_pipe_inode()]

```
476  static struct inode * get_pipe_inode(void)
477  {
478      struct inode *inode = get_empty_inode();
479
480      if (!inode)
481          goto fail_inode;
482
483      if(!pipe_new(inode))
484          goto fail_iput;
485      PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 1;
486      inode->i_fop = &rdwr_pipe_fops;
487      inode->i_sb = pipe_mnt->mnt_sb;
488
489      /*
490       * Mark the inode dirty from the very beginning,
491       * that way it will never be moved to the dirty
492       * list because "mark_inode_dirty()" will think
493       * that it already _is_ on the dirty list.
494       */
495      inode->i_state = I_DIRTY;
496      inode->i_mode = S_IFIFO | S_IRUSR | S_IWUSR;
497      inode->i_uid = current->fsuid;
498      inode->i_gid = current->fsgid;
499      inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
500      inode->i_blksize = PAGE_SIZE;
501      return inode;
502
503  fail_iput:
504      iput(inode);
505  fail_inode:
506      return NULL;
507  }
```

先在 478 行分配一个空闲的 inode 数据结构。这是一个比较复杂的数据结构，我们已在“文件系统 ”一章中加以详细介绍。对于管道的创建和使用，我们关心的只是其中少数几个成分。第一个成分 i_pipe 是指向一个 pipe_inode_info 数据结构的指针，只有当由一个 inode 数据结构所代表的文件是用来实现一个管道的时候才使用它，否则就把这个指针设为 NULL。pipe_inode_info 的数据结构是在 include/linux/pipe_fs_i.h 中定义的：

==================== include/linux/pipe_fs_i.h 5 15 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
5    struct pipe_inode_info {
6         wait_queue_head_t wait;
7         char *base;
8         unsigned int start;
9         unsigned int readers;
10        unsigned int writers;
11        unsigned int waiting_readers;
12        unsigned int waiting_writers;
13        unsigned int r_counter;
14        unsigned int w_counter;
15   };
```

同一文件中还定义了一些宏操作，下面我们就要用到这些宏定义。

==================== include/linux/pipe_fs_i.h 17 38 ====================

```
17   /* Differs from PIPE_BUF in that PIPE_SIZE is the length of the actual
18      memory allocation, whereas PIPE_BUF makes atomicity guarantees.  */
19   #define PIPE_SIZE      PAGE_SIZE
20
21   #define PIPE_SEM(inode)     (&(inode).i_sem)
22   #define PIPE_WAIT(inode)    (&(inode).i_pipe->wait)
23   #define PIPE_BASE(inode)    ((inode).i_pipe->base)
24   #define PIPE_START(inode)   ((inode).i_pipe->start)
25   #define PIPE_LEN(inode)     ((inode).i_size)
26   #define PIPE_READERS(inode) ((inode).i_pipe->readers)
27   #define PIPE_WRITERS(inode) ((inode).i_pipe->writers)
28   #define PIPE_WAITING_READERS(inode)  ((inode).i_pipe->waiting_readers)
29   #define PIPE_WAITING_WRITERS(inode)  ((inode).i_pipe->waiting_writers)
30   #define PIPE_RCOUNTER(inode)     ((inode).i_pipe->r_counter)
31   #define PIPE_WCOUNTER(inode)     ((inode).i_pipe->w_counter)
32
33   #define PIPE_EMPTY(inode)   (PIPE_LEN(inode) == 0)
34   #define PIPE_FULL(inode)    (PIPE_LEN(inode) == PIPE_SIZE)
35   #define PIPE_FREE(inode)    (PIPE_SIZE - PIPE_LEN(inode))
36   #define PIPE_END(inode)((PIPE_START(inode) + PIPE_LEN(inode)) & (PIPE_SIZE-1))
37   #define PIPE_MAX_RCHUNK(inode)   (PIPE_SIZE - PIPE_START(inode))
38   #define PIPE_MAX_WCHUNK(inode)   (PIPE_SIZE - PIPE_END(inode))
```

前面 get_pipe_inode()的代码中就引用了 PIPE_READERS 和 PIPE_WRITERS。分配了 inode 数据结构以后，483 行又通过 pipe_new()分配所需的缓冲区。这个函数的代码在 fs/pipe.c 中：

==================== fs/pipe.c 442 465 ====================

[sys_pipe()>do_pipe()>get_pipe_inode()>pipe_new()]

```
442  struct inode* pipe_new(struct inode* inode)
443  {
444       unsigned long page;
445
446       page = __get_free_page(GFP_USER);
447       if (!page)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
448             return NULL;
449
450         inode->i_pipe = kmalloc(sizeof(struct pipe_inode_info), GFP_KERNEL);
451         if (!inode->i_pipe)
452             goto fail_page;
453
454         init_waitqueue_head(PIPE_WAIT(*inode));
455         PIPE_BASE(*inode) = (char*) page;
456         PIPE_START(*inode) = PIPE_LEN(*inode) = 0;
457         PIPE_READERS(*inode) = PIPE_WRITERS(*inode) = 0;
458         PIPE_WAITING_READERS(*inode) = PIPE_WAITING_WRITERS(*inode) = 0;
459         PIPE_RCOUNTER(*inode) = PIPE_WCOUNTER(*inode) = 1;
460
461         return inode;
462     fail_page:
463         free_page(page);
464         return NULL;
465     }
```

这里先分配一个内存页面用作管道的缓冲区，再分配一个缓冲区用作 pipe_inode_info 数据结构。前面讲过，用来实现管道的文件是无形的，它并不出现在磁盘或其它的文件系统存储介质上，而只存在于内存空间，其它进程也无从“打开”或访问这个文件。所以，这个所谓文件实质上只是一个用作缓冲区的内存页面，只是把它纳入了文件系统的机制，借用了文件系统的各种数据结构和操作加以管理而已。

在前一章中已经讲过，inode 数据结构中有个重要的成分 i_fop，是指向一个 file_operations 数据结构的指针。在这个数据结构中给出了用于该文件的每种操作的函数指针。对于管道（见上面 get_pipe_inode()中的 486 行），这个数据结构是 rdwr_pipe_fops，也是在 fs/pipe.c 中定义的：

==================== fs/pipe.c 432 440 ====================

```
432     struct file_operations rdwr_pipe_fops = {
433         llseek:     pipe_lseek,
434         read:       pipe_read,
435         write:      pipe_write,
436         poll:       pipe_poll,
437         ioctl:      pipe_ioctl,
438         open:       pipe_rdwr_open,
439         release:  pipe_rdwr_release,
440     };
```

结合上述 include/linux/pipe_fs_i.h 中的宏定义，读者应该不难理解 get_pipe_inode()中自分配了 inode 结构以后的一些初始化操作，我们就不多讲了。值得注意的是，代码中并没有设置 inode 结构中的 inode_operations 结构指针 i_op，所以该指针为 0。可见，对于用来实现管道的 inode 并不允许对这里的 inode 进行常规操作，只有当 inode 代表着“有形”的文件时才使用。

从 get_pipe_inode()返回到 do_pipe()中时，必需的数据结构都已经齐全了。但是，还要为代表着管道两端的两个已打开文件分别分配“打开文件号”，这是通过调用 get_unused_fd()完成的。

让我们在 do_pipe()中继续往下看（fs/pipe.c）：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/pipe.c 542 572 ====================

[sys_pipe()>do_pipe()]

```
542         error = -ENOMEM;
543         sprintf(name, "[%lu]", inode->i_ino);
544         this.name = name;
545         this.len = strlen(name);
546         this.hash = inode->i_ino; /* will go */
547         dentry = d_alloc(pipe_mnt->mnt_sb->s_root, &this);
548         if (!dentry)
549              goto close_f12_inode_i_j;
550         dentry->d_op = &pipefs_dentry_operations;
551         d_add(dentry, inode);
552         f1->f_vfsmnt = f2->f_vfsmnt = mntget(mntget(pipe_mnt));
553         f1->f_dentry = f2->f_dentry = dget(dentry);
554 
555         /* read file */
556         f1->f_pos = f2->f_pos = 0;
557         f1->f_flags = O_RDONLY;
558         f1->f_op = &read_pipe_fops;
559         f1->f_mode = 1;
560         f1->f_version = 0;
561 
562         /* write file */
563         f2->f_flags = O_WRONLY;
564         f2->f_op = &write_pipe_fops;
565         f2->f_mode = 2;
566         f2->f_version = 0;
567 
568         fd_install(i, f1);
569         fd_install(j, f2);
570         fd[0] = i;
571         fd[1] = j;
572         return 0;
```

在正常的情况下，每个文件都至少有一个“目录项”，代表这个文件的一个路径名；而每个目录项则只描述一个文件，在 dentry 数据结构中有个指针指向相应的 inode 结构。因此，在 file 数据结构中有个指针 f_dentry 指向所打开文件的目录项 dentry 数据结构，这样，从 file 结构开始就可以一路通到文件的 inode 结构。对于管道来说，由于文件是无形的，本来并不非得有个目录项不可。可是，在 file 数据结构中并没有直接指向相应 inode 结构的指针，一定要经过一个目录项中转一下才行。而 inode 结构又是各种文件操作的枢纽。这么一来，对于管道就也得有一个目录项了。所以代码中的 547 行调用 d_alloc()分配一个目录项，然后通过 d_add()使已经分配的 inode 结构与这个目录项互相挂上钩，并且让两个已打开文件结构中的 f_dentry 指针都指向这个目录项。

对目录项的操作是通过一个 dentry_operations 数据结构定义的。具体到管道文件的目录项，这个数据结构是 pipefs_dentry_operations，这是在 550 行中设置的，定义于 fs/pipe.c 中：

==================== fs/pipe.c 472 474 ====================

```
static struct dentry_operations pipefs_dentry_operations = {
```

```
    d_delete:pipefs_delete_dentry,
};
```

就是说，对于管道的目录项只允许一种操作，那就是 pipefs_delete_dentry()，即把它删除。

对于管道的两端来说，管道是单向的，所以其 f1 一端设置成“只读”（O_RDONLY），而另一端则设置成“只写”（O_WRONLY）。同时，两端的文件操作也分别设置成 read_pipe_fops 和 write_pipe_fops，那都是在 fs/pipe.c 中定义的：

==================== fs/pipe.c 412 430 ====================

```
struct file_operations read_pipe_fops = {
    llseek:        pipe_lseek,
    read:          pipe_read,
    write:         bad_pipe_w,
    poll:          pipe_poll,
    ioctl:         pipe_ioctl,
    open:          pipe_read_open,
    release:  pipe_read_release,
};

struct file_operations write_pipe_fops = {
    llseek:        pipe_lseek,
    read:          bad_pipe_r,
    write:         pipe_write,
    poll:          pipe_poll,
    ioctl:         pipe_ioctl,
    open:          pipe_write_open,
    release:  pipe_write_release,
};
```

比较一下，就可以发现，在 read_pipe_fops 中的写操作函数为 bad_pipe_w()，而在 write_pipe_fops 中的读操作函数为 bad_pipe_r()，分别用来返回一个出错代码。读者可能会问，前面在管道的 inode 数据结构中将指针 i_fop 设置成指向 rdwr_pipe_fops，那显然是双向的，这里不是有矛盾吗？其实不然。对于代表着管道两端的两个已打开文件来说，一个只能写而另一个只能读，这是事情的一个方面。可是，另一方面，这两个逻辑上的已打开文件都通向同一个 inode、同一个物理上存在的“文件”，即用作管道的缓冲区；显然这个缓冲区应该既支持写又支持读，这才能使数据流通。读者在“文件系统”一章中看到，file 结构中的指针 f_op 一般都来自 inode 结构中的指针 i_fop，都指向同一个 file_operations 结构。而这里，对于管道这么一种特殊的文件，则使管道两端的 file 结构各自指向不同的 file_operations 结构，以此来确保一端只能读而另一端只能写。

管道是一种特殊的文件，它并不属于某种特定的文件系统（如 Ext2），而是自己构成一种独立的文件系统，也有自身的数据结构 pipe_fs_type（见 fs/pipe.c）：

==================== fs/pipe.c 632 633 ====================

```
static DECLARE_FSTYPE(pipe_fs_type, "pipefs", pipefs_read_super,
    FS_NOMOUNT|FS_SINGLE);
```

系统在初始化时通过 kern_mount()安装这个特殊的文件系统，并让一个指针 pipe_mnt 指向安装时

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

的 vfsmount 数据结构：

==================== fs/pipe.c 467 467 ====================
```
467  static struct vfsmount *pipe_mnt;
```

现在，代表着管道两端的两个文件既然都属于这个文件系统，它们各自的 file 结构中的指针 f_vfsmnt 就要指向安装该文件系统的 vfsmount 数据结构，而这个数据结构也就多了两个使用者，所以要调用 mntget()两次（见 552 行），使其使用计数加 2。

最后，do_pipe()中的 568 行和 569 行把两个已打开文件结构跟分配得的打开文件号挂起钩来（注意，打开文件号只在一个进程的范围内有效）；并且将两个打开文件号填入数组 fd[]中，使得 fd[0]为管道读出端的打开文件号，而 fd[1]为写入端的打开文件号。这个数组随后在 sys_pipe()中被复制到当前进程的用户空间。

显然，管道的两端在创建之初都在同一进程中，这样是起不到进程间通信的作用的。那么，怎样才能将管道用于进程间通信呢？下面就是一个典型的过程。

(1) 进程 A 创建一个管道，创建完成时代表管道两端的两个已打开文件都在进程 A 中。示意图如图 6.1。

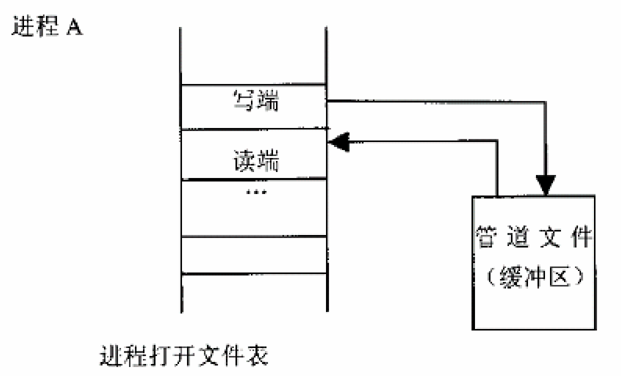


图 6.1 父进程创建管道

(2) 进程 A 通过 fork()创建出进程 B，在 fork()的过程中进程 A 的打开文件表按原样复制到进程 B 中。见图 6.2。
(3) 进程 A 关闭管道的读端，而进程 B 关闭管道的写端。于是，管道的写端在进程 A 中而读端在进程 B 中，成为父子进程之间的通信管道，见图 6.3。
(4) 进程 A 又通过 fork()创建进程 C，然后关闭其管道写端而与管道脱离关系，使得管道的写端在进程 C 中而读端在进程 B 中，成为两个兄弟进程之间的管道，如图 6.4。
(5) 进程 C 和进程 B 各自通过 exec()执行各自的目标程序，并通过管道进行单向通信。

如果考虑一个使用管道的 shell 命令行：

“ls -l | wc -l”

则上面的进程 A 相当于 shell，进程 B 执行 “ wc -l”，而进程 C 执行 “ ls -l”。不过，在进程 C 中要

PDF-XCHANGE Click to buy NOW! www.docu-track.com

将“标准输出通道”stdout 重定向到管道的写端，而在进程 B 中则要将“标准输入通道”stdin 重定向到管道的读端。

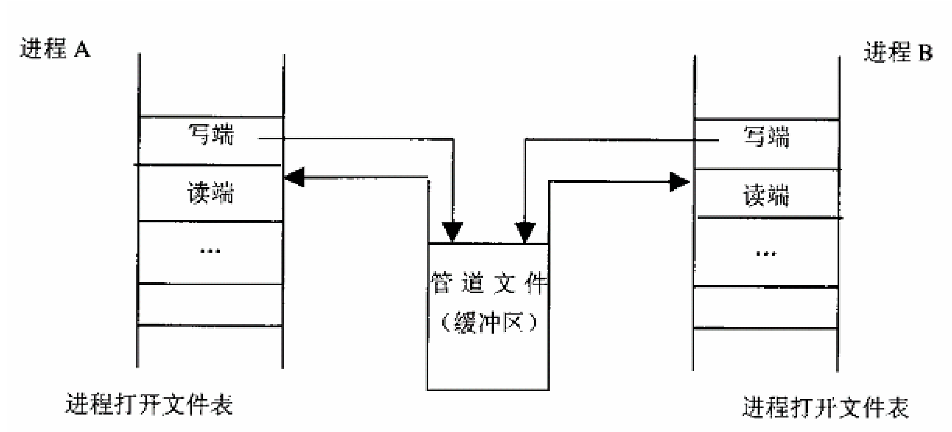


图 6.2　父子进程共享管道

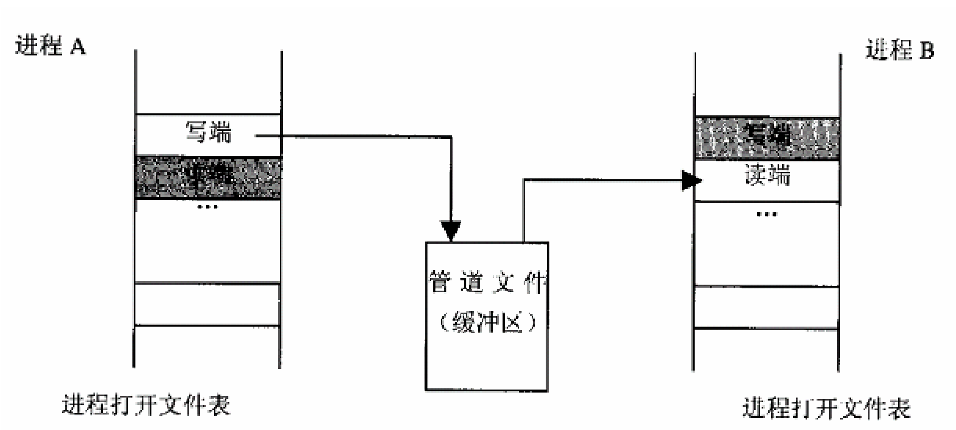


图 6.3　父子进程通过管道单向通讯

PDF-XCHANGE Click to buy NOW! www.docu-track.com

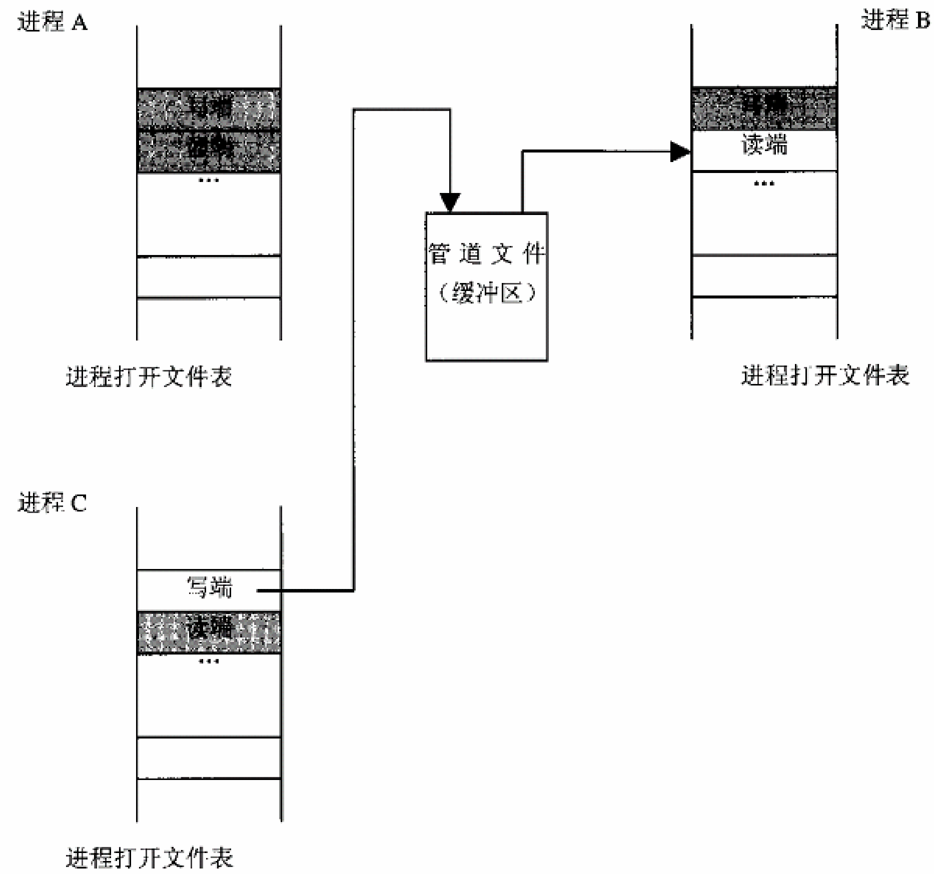


图 6.4　兄弟进程通过管道单向通讯

下面我们看一个实例：

```
#include <stdio.h>

int main( )
{
     int child_B, child_C;
     int pipefds[2];  /* pipefds[0] for read, pipefds[1] for write */
     char *args1[ ] = {"/bin/wc", NULL};
     char *args2[ ] = {"/usr/bin/ls", "-l", NULL};

     /* process A */
     pipe(pipefds); /* create a pipe */

     if (!(child_B = fork( )))  /* fork process B */
     {
          /**** Process B ****/
          close(pipefds[1]);  /* close the write end */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /* redirect stdin */
        close(0);
        dup2(pipefds[0],  0);
        close(pipefds[0]);
        /* exec the target */
        execve("/usr/bin/wc", args1, NULL); /* no return if success */
        printf("pid %d: I am back, something is wrong!\n", getpid());
    }
    /* process A continues */
    close(pipefds[0]); /* close the read end */
    if (!(child_C = fork())) /* fork process C */
    {
        /**** process C ****/
        /* redirect stdout */
        close(1) ;
        dup2(pipefds[1], 1);
        close(pipefds[1]);
        /* exec the target */
        execve("/bin/ls", args2, NULL); /* no return if success */
        printf("pid %d: I am back, something is wrong!\n", getpid( ));
    }

    /* process A continues */
    close(pipefds[1]); /* close the write end */
    wait4(child_B, NULL, 0, NULL); /* wait for process B to finish */
    printf("Done!\n") ;

    return 0;
}
```

程序中调用的 dup2()是一个系统调用，它将一个已打开文件的 file 结构指针复制到由指定的打开文件号所对应的通道。在进程 B 中，先把打开文件号 0（即标准输入）关闭，然后把管道的读端复制到文件号 0 所对应的通道，这就完成了对标准输入的重定向。但是原先的管道读端既然已经复制就没有用处了，所以也将其关闭。进程 C 的重定向与此相似，只不过所重定向的是标准输出。除此之外，就与前面所述的过程完全一样了。

从进程间通信的角度来说，这种标准输入和标准输出的重定向并非必须，直接使用管道原先的写端和读端也能在进程之间传递数据。但是，应该承认这种将标准输入 / 输出重定向与管道结合使用的办法是非常巧妙的，不这样就难以达到将可执行程序在启动执行时动态地加以组合的灵活性。另一方面，一旦将标准输入和标准输出分别重定向到管道的读端和写端以后，两个进程就都像对普通文件一样地读 / 写。事实上，它们并不知道自己在读 / 写的到底是一个文件，一个设备，还是一个管道。

但是，我们知道当读一个文件到达末尾的时候会碰到 EOF，从而知道已经读完了，可是当从管道中读时应该读到什么时候为止呢？下面读者将会看到，向管道写入的进程在完成其“使命”以后就会关闭管道的写端，一旦管道没有了写端就相当于到达了文件的末尾。

从管道所传递数据的角度看，两端的两个进程之间是一种典型的生产者 / 消费者关系。一旦生产者停止了生产并关闭了管道的写入端，消费者就没有东西可消费了，这时候就到了文件（管道）的末

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

尾。在典型的情况下，生产者总是在完成了使命，调用 exit()之前关闭其所有的已打开文件，包括管道，而消费者则总是在得知已经到达了输入文件末尾时才调用 exit()，所以一般总是生产者调用 exit()在前，消费者调用 exit()在后。但是，在特殊的条件下，也会有消费者 exit()在前的情况发生（例如消费者进程发生了非法越界访问），而使得管道的读端关闭在前。在这种情况下内核会向生产者进程发出一个 SIGPIPE 信号，表示管道已经“断裂”，而生产者进程在接收到这种信号时通常就会调用 exit()。

下面，我们进一步看看对管道的关闭以及读、写操作的源代码，以加深对管道机制的了解和理解。

先看管道的关闭。当一个进程通过系统调用 close()关闭代表着管道一端的已打开文件时，在内核中经由如下的执行路线而到达 fput()：

sys_close()>filp_close()>fput()

关于这条执行路线的详情可参阅“文件系统”一章。每当对一个已打开文件执行“关闭”操作时，在 fput()中将相应 file 数据结构中的共享计数减 1。如果减 1 以后该共享计数变成了 0，就进而通过具体文件系统提供的 release 操作，释放对 dentry 数据结构的使用，并释放 file 数据结构。

在最初打开一个文件时，内核为之分配一个 file 数据结构，并将共享计数设置成 1。那么，在什么情况下这个共享计数会变成大于 1，从而使得在一次调用 fput()后共享计数不变成 0 呢？

第一种情况是在 fork()一个进程的时候，读者在第 4 章已经看到过在 do_fork()中要调用一个函数 copy_files()；里面有个 for 循环，对所有已打开文件的 file 结构调用 get_file()递增其共享计数。所以，在 fork()出来一个子进程以后，若父进程先关闭一个已打开文件，便只会使其共享计数减 1，而并不会使计数到达 0，因而也就不会最终地关闭文件。

第二种情况是通过系统调用 dup()或 dup2()“复制”一个打开文件号。这与将同一个文件再打开一次是不同的，它只是使一个已经存在的 file 数据结构和本进程的另一个打开文件号建立联系而已。因此，前面所举的例子中将标准输入重定向到一个管道时，先是 dup2()然后 close()，并不会使其 file 结构中的共享计数变成 0。

也就是说，只有在一个 file 结构的最后一个“用户”通向该结构的最后一条通路也被关闭时，才会调用具体文件系统提供的 release 操作并最终释放 file 数据结构。

函数 fput()所处理的对象是与所关闭的文件相联系的 dentry 等数据结构。在 do_pipe()的代码中，我们已经看到管道两端的文件操作结构（file_operations 结构）被分别设置成 read_pipe_fops 和 write_pipe_fops。两个数据结构中对应于 release 的函数指针分别为 pipe_read_release 和 pipe_write_release()。所以，在 fput()采用这些指针来调用相应的函数时就会执行 pipe_read_release()或 pipe_write_release()。这两个函数都是通向 pipe_release()的“中转站”，或者说是 pipe_release()的“外层”，继续沿着 fs/pipe.c 往下看：

==================== fs/pipe.c 321 331 ====================

[sys_close()>filp_close()>fput()>pipe_read_release()]

```
static int
pipe_read_release(struct inode *inode, struct file *filp)
{
	return pipe_release(inode, 1, 0);
}

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static int
pipe_write_release(struct inode *inode, struct file *filp)
{
    return pipe_release(inode, 0, 1);
}
```

其主体为函数 pipe_release()，源代码在 fs/pipe.c 中。

结合这两个函数以及前面所列的 pipe_fs_i.h 中的一些宏定义，pipe_release()的代码就不难读懂了。

==================== fs/pipe.c 302 319 ====================

[sys_close()>filp_close()>fput()>pipe_read_release()>pipe_release()]

```
static int
pipe_release(struct inode *inode, int decr, int decw)
{
    down(PIPE_SEM(*inode));
    PIPE_READERS(*inode) -= decr;
    PIPE_WRITERS(*inode) -= decw;
    if (!PIPE_READERS(*inode) && !PIPE_WRITERS(*inode)) {
        struct pipe_inode_info *info = inode->i_pipe;
        inode->i_pipe = NULL;
        free_page((unsigned long) info->base);
        kfree(info);
    } else {
        wake_up_interruptible(PIPE_WAIT(*inode));
    }
    up(PIPE_SEM(*inode));

    return 0;
}
```

就像在 file 结构中有共享计数一样，在由 inode->i_pipe 所指向的 pipe_inode_info 结构中也有共享计数，而且有两个，一个是 readers，一个是 writers。这两个共享计数在创建管道时在 get_pipe_inode()中都被设置成 1（见 fs/pipe.c：get_pipe_inode()中的 485 行）。然后，当关闭管道的读端时，pipe_read_release()调用 pipe_release()，使共享计数 readers 减 1；而关闭管道的写端时则使 writers 减 1。当二者都减到了 0 时，整个管道就完成了使命，此时应将用作缓冲区的存储页面以及 pipe_inode_info 数据结构释放。在常规的文件操作中，文件的 inode 存在于磁盘（或其它介质）上，所以在最后关闭时要将内存中的 inode 数据结构写回到磁盘上。但是，管道并不是常规意义上的文件，磁盘上并没有相应的索引节点，所以最后只是将分配的 inode 数据结构（以及 dentry 结构）释放了事，而并没有什么磁盘操作。这一点从用于管道的 inode 数据结构中的 inode_operations 结构指针为 0 可以看出。

再看管道的读操作。在典型的应用中，管道的读端总是处于一个循环中，通过系统调用 read()从管道中读，读了就处理，处理完又读。对管道的读操作，在内核中经过 sys_read()和数据结构 read_pipe_fops 中的函数指针而到达 pipe_read()。这个函数的代码在 fs/pipe.c 中，让我们逐段地往下看：

==================== fs/pipe.c 38 49 ====================

[sys_read()>pipe_read()]

```
static ssize_t
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
39  pipe_read(struct file *filp, char *buf, size_t count, loff_t *ppos)
40  {
41       struct inode *inode = filp->f_dentry->d_inode;
42       ssize_t size, read, ret;
43
44       /* Seeks are not allowed on pipes.  */
45       ret = -ESPIPE;
46       read = 0;
47       if (ppos != &filp->f_pos)
48            goto out_nolock;
49
```

这里 44 行的注解说 seek 操作在管道上是不允许的，这当然是对的，事实上函数 pipe_lseek()只是返回一个出错代码。注意 47 行的检验所针对的只是参数 ppos，那是个指针，必须指向 filp->f_pos 本身。沿着 fs/pipe.c 再往下看：

==================== fs/pipe.c 50 84 ====================

[sys_read()>pipe_read()]

```
50       /* Always return 0 on null read.  */
51       ret = 0;
52       if (count == 0)
53            goto out_nolock;
54
55       /* Get the pipe semaphore */
56       ret = -ERESTARTSYS;
57       if (down_interruptible(PIPE_SEM(*inode)))
58            goto out_nolock;
59
60       if (PIPE_EMPTY(*inode)) {
61  do_more_read:
62            ret = 0;
63            if (!PIPE_WRITERS(*inode))
64                 goto out;
65
66            ret = -EAGAIN;
67            if (filp->f_flags & O_NONBLOCK)
68                 goto out;
69
70            for (;;) {
71                 PIPE_WAITING_READERS(*inode)++;
72                 pipe_wait(inode);
73                 PIPE_WAITING_READERS(*inode)--;
74                 ret = -ERESTARTSYS;
75                 if (signal_pending(current))
76                      goto out;
77                 ret = 0;
78                 if (!PIPE_EMPTY(*inode))
79                      break;
80                 if (!PIPE_WRITERS(*inode))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
81                      goto out;
82              }
83          }
84
```

如果读的时候管道里已经有数据在缓冲区中，这一段程序就被跳过了。可是，如果管道缓冲区中没有数据，那一般就要睡眠等待了，但是有两种例外的情况。第一种情况是管道的 writers 计数已经为 0，也就是说已经没有“生产者”会向管道中写了，这时候当然不能再等待。第二种情况是管道创建时设置了标志位 O_NOBLOCK，表示在读不到东西时，当前进程不应被“阻塞”（也就是在睡眠中等待），而要立即返回。只要不属于这两种特殊情况，那就要通过 pipe_wait()在睡眠中等待了。函数 pipe_wait()的代码也在同一文件 fs/pipe.c 中：

==================== fs/pipe.c 25 36 ====================

[sys_read()>pipe_read()>pipe_wait()]

```
25    /* Drop the inode semaphore and wait for a pipe event, atomically */
26    void pipe_wait(struct inode * inode)
27    {
28        DECLARE_WAITQUEUE(wait, current);
29        current->state = TASK_INTERRUPTIBLE;
30        add_wait_queue(PIPE_WAIT(*inode), &wait);
31        up(PIPE_SEM(*inode));
32        schedule();
33        remove_wait_queue(PIPE_WAIT(*inode), &wait);
34        current->state = TASK_RUNNING;
35        down(PIPE_SEM(*inode));
36    }
```

有关信号量和等待队列的使用以及进程调度请参看第 4 章。注意，与这里的 up()操作配对的 down_interruptible()，是在 pipe_read()代码中的 57 行，一个在 for 循环外面，一个在 for 循环里面。实际上，pipe_read()中的临界区是从 57 行至 127 行（见下面的代码），但是睡眠时必须要退出临界区，而到被唤醒后再进入临界区。为什么要把 pipe_wait()放在一个循环中呢？这是因为睡眠中的进程被唤醒的原因不一定就是有进程往管道中写，也可能是收到了信号。而且，即使是因为有进程往管道中写而唤醒，也不能保证每个被唤醒的进程都能读到数据，因为等待着从管道中读数据的进程可能不止一个。因此，要将睡眠等待的过程放在一个循环中，并且在唤醒以后还要再检验所等待的条件是否得到满足，以及是否发生了例外的情况。对于在生产者 / 消费者模型中消费者一方的等待过程，这是一种典型的设计。在正常的情况下，这个循环一般都是因为管道中有了数据而结束（见 78 和 79 行），于是具体从管道中读取数据的操作就开始了（fs/pipe.c）：

==================== fs/pipe.c 85 132 ====================

[sys_read()>pipe_read()]

```
85        /* Read what data is available.  */
86        ret = -EFAULT;
87        while (count > 0 && (size = PIPE_LEN(*inode))) {
88            char *pipebuf = PIPE_BASE(*inode) + PIPE_START(*inode);
89            ssize_t chars = PIPE_MAX_RCHUNK(*inode);
90
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (chars > count)
            chars = count;
        if (chars > size)
            chars = size;

        if (copy_to_user(buf, pipebuf, chars))
            goto out;

        read += chars;
        PIPE_START(*inode) += chars;
        PIPE_START(*inode) &= (PIPE_SIZE - 1);
        PIPE_LEN(*inode) -= chars;
        count -= chars;
        buf += chars;
    }

    /* Cache behaviour optimization */
    if (!PIPE_LEN(*inode))
        PIPE_START(*inode) = 0;

    if (count && PIPE_WAITING_WRITERS(*inode) && !(filp->f_flags & O_NONBLOCK)) {
        /*
         * We know that we are going to sleep: signal
         * writers synchronously that there is more
         * room.
         */
        wake_up_interruptible_sync(PIPE_WAIT(*inode));
        if (!PIPE_EMPTY(*inode))
            BUG();
        goto do_more_read;
    }
    /* Signal writers asynchronously that there is more room.  */
    wake_up_interruptible(PIPE_WAIT(*inode));

    ret = read;
out:
    up(PIPE_SEM(*inode));
out_nolock:
    if (read)
        ret = read;
    return ret;
}
```

每个管道只有一个页面用作缓冲区，该页面是按环形缓冲区的方式来使用的。就是说，每当读 / 写到了页面的末端就又要回到页面的始端（见图 6.5），这样，管道中的数据就有可能要分两段读出，所以要由一个循环来完成。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

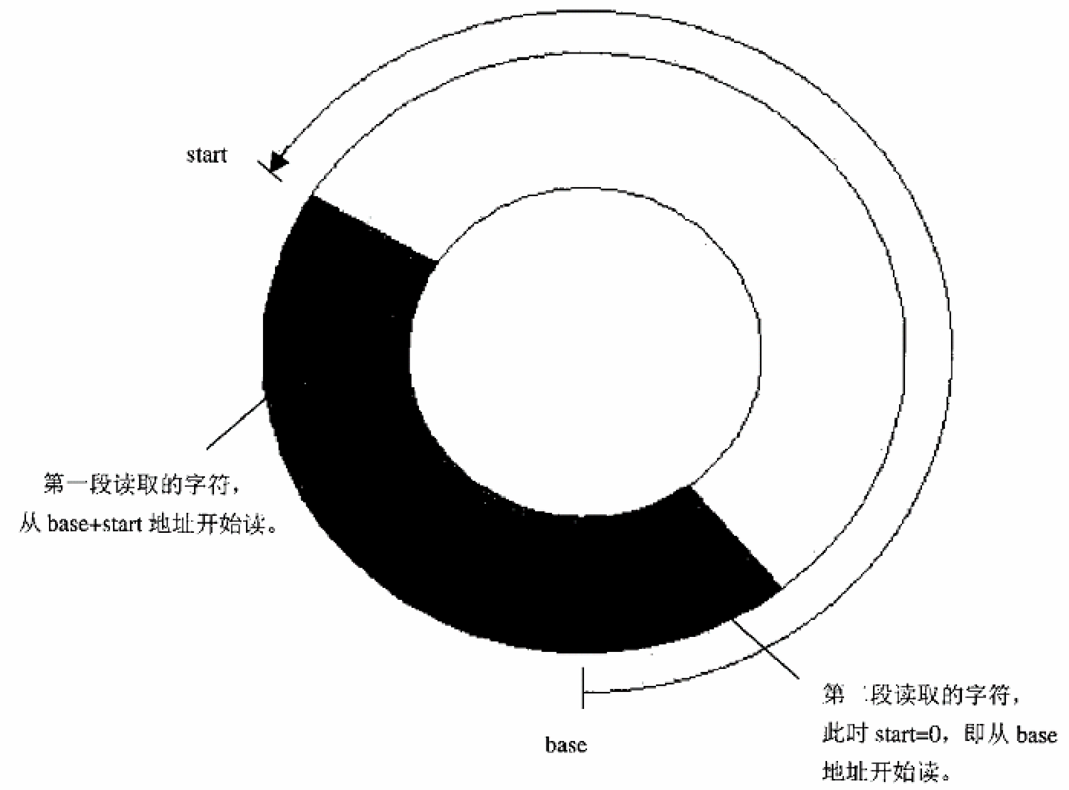


图 6.5　管道的环形缓冲区示意图

结合本节前面所列的宏定义，这段代码应该是不难理解的。循环结束以后的情况有以下几种可能：

(1) 读到了所要求的长度，所以 count 减到了 0，同时管道中的数据也正好读完了，所以管道中的数据长度变成了 0。此时函数的返回值为所要求的长度。

(2) 管道中的数据已经读完，但还没有达到所要求的长度，函数返回实际读出的长度。

(3) 读到了所要求的长度，但管道中的数据还有剩余，此时函数也是返回所要求的长度。

在前两种情况下，管道中的数据都已读完，但指示着下一次读 / 写的起始点 start，在不同的条件下有可能在页面中的任何位置上。可是，既然管道中已经空了，那就不如把起始点 start 设置到页面的开头，这样可以减少下一次读 / 写必须分成两段进行的可能性，这就是 108 行和 109 行所作优化的目的。

由于管道的缓冲区只限于一个页面，当“生产者”进程有大量数据要写时，每当写满了一个页面（分一段或两段）就得停下来睡眠等待，等到消费者进程从管道中读走了一些数据而腾出一些空间时才能继续。所以，“消费者”进程在读出了一些数据以后要唤醒可能正在睡眠中的“生产者”进程。最后，只要读出的长度不为 0，就要更新 inode 的受访问时间印记。

所有这些操作，包括从管道中读出，复制到用户空间，更新 inode 的受访问时间印记等等，都是不能容许其它进程打扰的，所以都是放在临界区中进行。而 57 行处的 down_interruptible()和 127 行处的

up()正是界定了这样一个临界区。

与读操作相似，对管道的写操作也是在 sys_write()中通过 file 结构中的指针 f_op 找到 file_operations 数据结构 write_pipe_fops，再通过其函数指针 write 调用 pipe_write()。这个函数也是在 fs/pipe.c 中定义的，我们还是逐段来解读：

==================== fs/pipe.c 134 150 ====================

[sys_write()>pipe_write()]

```
static ssize_t
pipe_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
{
	struct inode *inode = filp->f_dentry->d_inode;
	ssize_t free, written, ret;

	/* Seeks are not allowed on pipes.  */
	ret = -ESPIPE;
	written = 0;
	if (ppos != &filp->f_pos)
		goto out_nolock;

	/* Null write succeeds.  */
	ret = 0;
	if (count == 0)
		goto out_nolock;

```

显然，这一段与 pipe_read()的开头一段完全相同。继续往下读：

==================== fs/pipe.c 151 180 ====================

```
	ret = -ERESTARTSYS;
	if (down_interruptible(PIPE_SEM(*inode)))
		goto out_nolock;

	/* No readers yields SIGPIPE.  */
	if (!PIPE_READERS(*inode))
		goto sigpipe;

	/* If count <= PIPE_BUF, we have to make it atomic.  */
	free = (count <= PIPE_BUF ? count : 1);

	/* Wait, or check for, available space.  */
	if (filp->f_flags & O_NONBLOCK) {
		ret = -EAGAIN;
		if (PIPE_FREE(*inode) < free)
			goto out;
	} else {
		while (PIPE_FREE(*inode) < free) {
			PIPE_WAITING_WRITERS(*inode)++;
			pipe_wait(inode);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
171                 PIPE_WAITING_WRITERS(*inode)--;
172                 ret = -ERESTARTSYS;
173                 if (signal_pending(current))
174                     goto out;
175
176                 if (!PIPE_READERS(*inode))
177                     goto sigpipe;
178             }
179         }
180
```

如果管道的读端已经全部关闭，那就表示已经没有“消费者”了。既然没有了“消费者”，那么“生产者”的存在以及继续“生产”就都失去了意义；所以此时转到标号 sigpipe 处，向当前进程发送一个 SIGPIPE 信号，表示管道已经“断裂”：

==================== fs/pipe.c 240 245 ====================

```
240  sigpipe:
241      if (written)
242          goto out;
243      up(PIPE_SEM(*inode));
244      send_sig(SIGPIPE, current, 0);
245      return -EPIPE;
```

一般而言，进程在接收到 SIGPIPE 信号时会调用 do_exit()来结束其生命。读者也许注意到，这里其实是当前进程自己向自己发 SIGPIPE 信号。那么为何不直接调用 do_exit()呢？这里有两方面的考虑：一方面是使程序的结构更好，更整齐划一；另一方面也为进程通过信号机制来改变其在接收到 SIGPIPE 信号时的行为提供了更多的灵活性甚至可能性。

160 行中的常数 PIPE_BUF 在 include/linux/limits.h 中定义为 4096。当要求写入的长度不超过这个数值时，内核保证写入操作的“原子性”，也就是说，一定要到管道缓冲区中足够容纳这块数据时才开始写。如果超过这个数值，就不保证其“原子性”了，这时候有多大空间就写多少字节，有一个字节的空间就写一个字节，余下的等“消费者”读走一些字节以后再继续写。这就是第 160 行将变量 free 设置成 count 或者 1 的意义。注意变量 free 表示开始写入前缓冲区中至少要有这么多个空闲的字节，否则就要睡眠等待；所以只是在决定等待与否时使用，而一旦开始写入就不再使用了。读者可以对照前面 pipe_read()中的代码自行阅读这里的 162～179 行，应该不会有困难。

一旦“生产者”进程在等待中被“消费者”进程唤醒，并且缓冲区中有了足够的空间，或者一开始时缓冲区中就有足够的空间，具体的写入操作就开始了（见 fs/pipe.c）：

==================== fs/pipe.c 181 238 ====================

[sys_write()>pipe_write()]

```
181      /* Copy into available space.  */
182      ret = -EFAULT;
183      while (count > 0) {
184          int space;
185          char *pipebuf = PIPE_BASE(*inode) + PIPE_END(*inode);
186          ssize_t chars = PIPE_MAX_WCHUNK(*inode);
187
```

```
        if ((space = PIPE_FREE(*inode)) != 0) {
            if (chars > count)
                chars = count;
            if (chars > space)
                chars = space;

            if (copy_from_user(pipebuf, buf, chars))
                goto out;

            written += chars;
            PIPE_LEN(*inode) += chars;
            count -= chars;
            buf += chars;
            space = PIPE_FREE(*inode);
            continue;
        }

        ret = written;
        if (filp->f_flags & O_NONBLOCK)
            break;

        do {
            /*
             * Synchronous wake-up: it knows that this process
             * is going to give up this CPU, so it doesnt have
             * to do idle reschedules.
             */
            wake_up_interruptible_sync(PIPE_WAIT(*inode));
            PIPE_WAITING_WRITERS(*inode)++;
            pipe_wait(inode);
            PIPE_WAITING_WRITERS(*inode)--;
            if (signal_pending(current))
                goto out;
            if (!PIPE_READERS(*inode))
                goto sigpipe;
        } while (!PIPE_FREE(*inode));
        ret = -EFAULT;
    }

    /* Signal readers asynchronously that there is more data.  */
    wake_up_interruptible(PIPE_WAIT(*inode));

    inode->i_ctime = inode->i_mtime = CURRENT_TIME;
    mark_inode_dirty(inode);

out:
    up(PIPE_SEM(*inode));
out_nolock:
    if (written)
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
237            ret = written;
     . . . . . . .
```

==================== fs/pipe.c 246 246 ====================

```
246  }
```

首先，对照 pipe_read()中分两段读的情况，即使要求写入的长度小于 PIPE_BUF 时，也可能会要分两段来写，所以整个写入的过程也放在一个 while 循环中。另外，要求写入的长度大于 PIPE_BUF 时，还要分成几次来写，也就是先写入若干字节，然后睡眠等待“消费者”从缓冲区中读走一些字节而创造出一些空间，再继续写。这就是为什么要有 209～223 行的 do-while 循环的原因。这个循环与前面的睡眠等待循环略有不同，这就是当进程被唤醒时，只检验缓冲区中是否有空间，而不问空间多大。为什么呢？因为此时的宗旨是有一个字节的空间就写一个字节，而既然“消费者”进程已经读走了若干字节，那么至少已经有一个字节的空间，可以进入 while 循环体的下一次循环了。对照 pipe_read()的代码，读者应该可以读懂上面这段代码而不会有太大的困难，我们把它留给读者作练习。建议读者假设几种不同的数据长度来走过这段程序，并且在纸上记下几种不同情况下的执行路线。阅读时要注意 202 行的 continue 语句，当要求写入的数据长度不大于 PIPE_BUF 但需要分两段（不是两次）写入时，它使执行路线跳过后面的 do-while 循环。同时，还要注意 185 行中的宏定义 PIPE_END()，它使写入的位置 pipe_buf 回到页面的起点。

这样，在典型的情景下，“生产者”和“消费者”之间互相等待，互相唤醒，协调地向前发展，也就是说：

- 对“生产者”而言，缓冲区中有空间就往里写，并且唤醒可能正在等待着要从缓冲区中读数据的“消费者”；没有空间就睡眠，等待“消费者”从缓冲区读走数据而腾出空间（然后被唤醒）。
- 对“消费者”而言，缓冲区中有数据就读出，然后唤醒可能正在等待着要往缓冲区写的“生产者”。如没有数据就睡眠，等待“生产者”往缓冲区中写数据（然后被唤醒）。

一句话，管道两端的进程通过管道所形成的是典型的“生产者 / 消费者”关系和运行模式。

6.3 命名管道

应该说，前一节中的“管道”机制是一项重要的发明，它为 Unix 操作系统所带来的变化是革命性的，甚至可以说，没有管道就没有当初“ Unix 环境”的形成。但是，人们也认识到，管道机制也存在着一些缺点和不足。由于管道是一种“无名”、“无形”的文件，它就只能通过 fork()的过程创建于“近亲”的进程之间，而不可能成为可以在任意两个进程之间建立通信的机制，更不可能成为一种一般的、通用的进程间通信模型。同时，管道机制的这种缺点本身就强烈地暗示着人们，只要用“有名”、“有形”的文件来实现管道，就能克服这种缺点。这里所谓“有名”是指这样一个文件应该有个文件名，使得任何进程都可以通过文件名或路径名与这个文件挂上钩；所谓“有形”是指文件的 inode 应该存在于磁盘或其它文件系统介质上，使得任何进程在任何时间（而不仅仅是在 fork()时）都可以建立（或断开）与这个文件之间的联系。所以，有了管道以后，“命名管道”的出现就是必然的了。

为了实现“命名管道”，在“普通文件”、“块设备文件”、“字符设备文件”之外，又设立了一种文

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

件类型，称为 FIFO 文件（“先进先出”文件）。对这种文件的访问严格遵循“先进先出”的原则，而不允许有在文件内移动读写指针位置的 lseek()操作。这样一来，就可以像在磁盘上建立一个文件一样地建立一个命名管道，具体可以使用命令 mknod 来建立。例如：

%mknod mypipe p

这里的参数“ p”表示所建立的节点（也即特殊文件）的类型为命名管道。当然，也可以在程序中通过系统调用 mknod()来达到同样的目的，只不过此时在调用参数 mode 中要设置一个标志位 S_IFIFO，表示要创建的是一个 FIFO 文件。

建立了这样的节点以后，有关的进程就可以像打开一个文件一样地来“打开”与这个命名管道的联系。对 FIFO 文件上的操作由下列几个 file_operations 数据结构确定，这些代码都在 fs/pipe.c 中。

==================== fs/pipe.c 378 410 ====================

```
/*
 * The file_operations structs are not static because they
 * are also used in linux/fs/fifo.c to do operations on FIFOs.
 */
struct file_operations read_fifo_fops = {
	llseek:		pipe_lseek,
	read:		pipe_read,
	write:		bad_pipe_w,
	poll:		fifo_poll,
	ioctl:		pipe_ioctl,
	open:		pipe_read_open,
	release:	pipe_read_release,
};

struct file_operations write_fifo_fops = {
	llseek:		pipe_lseek,
	read:		bad_pipe_r,
	write:		pipe_write,
	poll:		fifo_poll,
	ioctl:		pipe_ioctl,
	open:		pipe_write_open,
	release:	pipe_write_release,
};

struct file_operations rdwr_fifo_fops = {
	llseek:		pipe_lseek,
	read:		pipe_read,
	write:		pipe_write,
	poll:		fifo_poll,
	ioctl:		pipe_ioctl,
	open:		pipe_rdwr_open,
	release:	pipe_rdwr_release,
};
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

对照一下用于普通管道的数据结构 read_pipe_fops，write_pipe_fops 以及 rdwr_pipe_fops（见上节），就可以看出它们几乎是完全一样的。fifo_poll()和 pipe_poll()都用于 select()系统调用，与通信机制本身并无多大关系，我们在这里并不关心。所不同的只是，对于普通管道虽然也定义了相当于 open()的操作 pipe_read_open()，pipe_write_open()和 pipe_rdwr_open()，但是这些函数实际上在典型的应用中是不使用的。如前节所述，普通管道是通过 do_pipe()建立，通过 fork()的过程伸展到两个进程之间的。对于父进程，在系统调用 pipe()以后就已打开，而对于子进程则是与生俱来的，所以都不再需要再来打开。而命名管道就不同了，参加通信的进程确实要通过调用这些函数来“打开”通向已经建立在文件系统中的 FIFO 文件的通道。

既然主要的不同之处仅在于“打开”的过程，我们就来看看，一个进程是怎样通过 open()系统调用来建立与一个已经创建的 FIFO 文件之间的联系的。读者在“文件系统”一章中看到，进程在内核中由 sys_open()进入 filp_open()，然后在 open_namei()中调用一个函数 path_walk()，根据文件的路径名在文件系统中找到代表这个文件的 inode。在将磁盘上的 inode 读入内存时，要根据文件的类型（FIFO 文件的 S_IFIFO 标志位为 1），将 inode 中的 i_op 指针和 i_fop 指针设置成指向相应的 inode_operations 数据结构和 file_operations 数据结构，但是对于像 FIFO 这样的特殊文件则调用 init_special_inode()来加以初始化。这段代码在 ext2_read_inode()中（fs/ext2/inode.c）：

==================== fs/ext2/inode.c 1059 1078 ====================

[sys_open()>filp_open()>open_namei()>path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()>ext2_read_inode()]

```
	if (inode->i_ino == EXT2_ACL_IDX_INO ||
	    inode->i_ino == EXT2_ACL_DATA_INO)
		/* Nothing to do */ ;
	else if (S_ISREG(inode->i_mode)) {
		inode->i_op = &ext2_file_inode_operations;
		inode->i_fop = &ext2_file_operations;
		inode->i_mapping->a_ops = &ext2_aops;
	} else if (S_ISDIR(inode->i_mode)) {
		inode->i_op = &ext2_dir_inode_operations;
		inode->i_fop = &ext2_dir_operations;
	} else if (S_ISLNK(inode->i_mode)) {
		if (!inode->i_blocks)
			inode->i_op = &ext2_fast_symlink_inode_operations;
		else {
			inode->i_op = &page_symlink_inode_operations;
			inode->i_mapping->a_ops = &ext2_aops;
		}
	} else 
		init_special_inode(inode, inode->i_mode,
				   le32_to_cpu(raw_inode->i_block[0]));
```

可见，只要文件的类型不是 ACL 索引或数据（均用于访问权限控制），不是普通文件，不是目录，不是符号连接，就属于特殊文件，就要通过 init_special_inode()来初始化其 inode 结构（fs/devices.c）。

==================== fs/devices.c 200 216 ====================

[sys_open()>filp_open()>open_namei()>path_walk()>real_lookup()>ext2_lookup()>iget()>get_new_inode()

>ext2_read_inode()>init_special_inode()]

```
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
	inode->i_mode = mode;
	if (S_ISCHR(mode)) {
		inode->i_fop = &def_chr_fops;
		inode->i_rdev = to_kdev_t(rdev);
	} else if (S_ISBLK(mode)) {
		inode->i_fop = &def_blk_fops;
		inode->i_rdev = to_kdev_t(rdev);
		inode->i_bdev = bdget(rdev);
	} else if (S_ISFIFO(mode))
		inode->i_fop = &def_fifo_fops;
	else if (S_ISSOCK(mode))
		inode->i_fop = &bad_sock_fops;
	else
		printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
}
```

显然，对于 FIFO 文件，其 inode 结构中的 inode_operations 结构指针 i_op 为 0（代码中并未设置），而 file_operations 结构指针 i_fop 则指向 def_fifo_fops，定义于 fs/fifo.c：

==================== fs/fifo.c 150 157 ====================

```
/*
 * Dummy default file-operations: the only thing this does
 * is contain the open that then fills in the correct operations
 * depending on the access mode of the file...
 */
struct file_operations def_fifo_fops = {
	open:		fifo_open,	/* will set read or write pipe_fops */
};
```

与前一节中 pipe 文件的 inode 结构作一比较，就可以看出对于 pipe 文件的 inode 结构并没有走过这么一个过程，与 init_special_inode()也毫无关系。这是因为 pipe 文件的 inode 结构不是通过 ext2_read_inode()从磁盘上读入，而是临时生成出来的（因而是无名、无形的）。

随后，在 dentry_open()中将 inode 结构中的这个 file_operations 结构指针复制到 file 数据结构中。这样，对于命名管道，在打开文件时经由数据结构 def_fifo_fops，就可以得到函数指针 fifo_open，从而进入函数 fifo_open()。有关这个过程的详情可参看“文件系统”一章，这里我们关心的是进入了 fifo_open()以后的操作。函数 fifo_open()的代码在 fs/fifo.c 中：

==================== fs/fifo.c 31 46 ====================

[sys_open()>filp_open()>dentry_open()>fifo_open()]

```
static int fifo_open(struct inode *inode, struct file *filp)
{
	int ret;

	ret = -ERESTARTSYS;
	lock_kernel();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
37        if (down_interruptible(PIPE_SEM(*inode)))
38             goto err_nolock_nocleanup;
39
40        if (!inode->i_pipe) {
41             ret = -ENOMEM;
42             if(!pipe_new(inode))
43                  goto err_nocleanup;
44        }
45        filp->f_version = 0;
46
```

首先，当首次打开这个 FIFO 文件的进程来到 fifo_open()时，该管道的缓冲页面尚未分配，所以在 42 行通过 pipe_new()分配所需的 pipe_inode_info 数据结构和缓冲页面。以后再来打开同一 FIFO 文件的进程就会跳过这一段。再往下看 fs/fifo.c：

==================== fs/fifo.c 47 148 ====================

```
47        switch (filp->f_mode) {
48        case 1:
49        /*
50         *  O_RDONLY
51         *  POSIX.1 says that O_NONBLOCK means return with the FIFO
52         *  opened, even when there is no process writing the FIFO.
53         */
54             filp->f_op = &read_fifo_fops;
55             PIPE_RCOUNTER(*inode)++;
56             if (PIPE_READERS(*inode)++ == 0)
57                  wake_up_partner(inode);
58
59             if (!PIPE_WRITERS(*inode)) {
60                  if ((filp->f_flags & O_NONBLOCK)) {
61                       /* suppress POLLHUP until we have
62                        * seen a writer */
63                       filp->f_version = PIPE_WCOUNTER(*inode);
64                  } else
65                  {
66                       wait_for_partner(inode, &PIPE_WCOUNTER(*inode));
67                       if(signal_pending(current))
68                            goto err_rd;
69                  }
70             }
71             break;
72
73        case 2:
74        /*
75         *  O_WRONLY
76         *  POSIX.1 says that O_NONBLOCK means return -1 with
77         *  errno=ENXIO when there is no process reading the FIFO.
78         */
79             ret = -ENXIO;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if ((filp->f_flags & O_NONBLOCK) && !PIPE_READERS(*inode))
            goto err;

        filp->f_op = &write_fifo_fops;
        PIPE_WCOUNTER(*inode)++;
        if (!PIPE_WRITERS(*inode)++)
            wake_up_partner(inode);

        if (!PIPE_READERS(*inode)) {
            wait_for_partner(inode, &PIPE_RCOUNTER(*inode));
            if (signal_pending(current))
                goto err_wr;
        }
        break;

    case 3:
    /*
     *  O_RDWR
     *  POSIX.1 leaves this case "undefined" when O_NONBLOCK is set.
     *  This implementation will NEVER block on a O_RDWR open, since
     *  the process can at least talk to itself.
     */
        filp->f_op = &rdwr_fifo_fops;

        PIPE_READERS(*inode)++;
        PIPE_WRITERS(*inode)++;
        PIPE_RCOUNTER(*inode)++;
        PIPE_WCOUNTER(*inode)++;
        if (PIPE_READERS(*inode) == 1 || PIPE_WRITERS(*inode) == 1)
            wake_up_partner(inode);
        break;

    default:
        ret = -EINVAL;
        goto err;
    }

    /* Ok! */
    up(PIPE_SEM(*inode));
    unlock_kernel();
    return 0;

err_rd:
    if (!--PIPE_READERS(*inode))
        wake_up_interruptible(PIPE_WAIT(*inode));
    ret = -ERESTARTSYS;
    goto err;

err_wr:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (!--PIPE_WRITERS(*inode))
        wake_up_interruptible(PIPE_WAIT(*inode));
    ret = -ERESTARTSYS;
    goto err;

err:
    if (!PIPE_READERS(*inode) && !PIPE_WRITERS(*inode)) {
        struct pipe_inode_info *info = inode->i_pipe;
        inode->i_pipe = NULL;
        free_page((unsigned long)info->base);
        kfree(info);
    }

err_nocleanup:
    up(PIPE_SEM(*inode));

err_nolock_nocleanup:
    unlock_kernel();
    return ret;
}
```

FIFO 文件可以按三种不同的模式打开，就是“只读”、“只写”以及“读写”。同时，在系统调用 open()中还有个参数 flags。如果 flags 中的标志位 O_NONBLOCK 为 1，就表示在打开的过程中即使某些条件得不到满足也不要睡眠等待，而应立即返回。在典型的应用中，像对普通管道一样，一个进程按“只读”模式打开命名管道，成为“消费者”；而另一个进程则按“只写”模式打开命名管道，成为“生产者”。可是，在普通管道的情况下，管道的两端是由同一进程在 do_pipe()中同时“打开”的，而在命名管道的情况下则管道的两端通常分别由两个进程先后打开，这就有了个“同步”的问题。除此之外，还有个不同，就是普通管道既然是“无名”、“无形”，一般就不会有另一个进程也来“打开”这个管道。而在命名管道的情况下，则任意一个进程都可以通过相同的路径名打开同一个 FIFO 文件。这些因素都使建立命名管道的过程比建立普通管道的过程要复杂一些。

先来看命名管道的读端，也就是按“只读”模式打开一个 FIFO 文件时（case 1）的几种情况：

(1) 如果管道的写端已经打开，那么读端的打开就完成了命名管道的建立过程。在这种情况下，写端的进程，也就是“生产者”进程，一般都是正在睡眠中，等待着命名管道建立过程的完成，所以要将其唤醒。然后，两个进程差不多同时返回到各自的用户空间，尔后就可以通过这命名管道进行通信了。

(2) 如果命名管道的写端尚未打开，而 flags 中的 O_NONBLOCK 标志位为 1，表示不应等待。此时读端虽已打开，但命名管道只是部分地建立了，而 O_NONBLOCK 标志的使用又要求系统调用不加等待立即返回，所以不作等待。不过，这里把读端 file 结构中的 f_version 字段设置成 PIPE_WCOUNTER(*inode)，即对本管道写端的计数。这与通过 pipe_poll()对命名管道的查询有关，而与读写无关。读者可以结合系统调用 select()（见第 8 章“字符设备驱动”）阅读其代码。

(3) 如果命名管道的写端尚未打开，而 flags 中的 O_NONBLOCK 标志位为 0。在这种情况下，读端的打开只是完成了命名管道建立过程的一半，所以“消费者”进程要通过 wait_for_partner()

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进入睡眠，等待某个“生产者”进程来打开命名管道的写端以完成其建立过程。

不管是哪一种情况下，读端 file 结构中的 file_operations 指针 f_op 都指向 read_fifo_fops，为随后的读操作作好了准备。

相应地，命名管道写端的打开（case 2）也有以下几种不同的情况：

(1) 如果命名管道读端已经打开，那么写端的打开就完成了建立命名管道的过程。在这种情况下位于命名管道读端的进程（即“消费者”进程）有可能正在睡眠中等待，所以，如果当前进程是第一次打开该命名管道写端的进程，就要负责将其唤醒（见第 85、86 行）。

(2) 如果命名管道读端尚未打开，而 flags 中的 O_NONBLOCK 标志位为 0。在这种情况下“生产者”进程要睡眠等待至某个“消费者”进程打开该命名管道的读端才能返回。

(3) 如果命名管道的读端尚未打开，而 flags 中的 O_NONBLOCK 标志位为 1。此时对命名管道写端的打开失败，所以要释放已经分配的各种资源而返回-1。

至于对 FIFO 文件的“读写”打开，则相当于由同一个进程同时打开了命名管道的两端，所以不管怎样都不需要等待。但是还有可能已经有某个进程先打开了写端或读端而正在睡眠中等待，所以只要有任何一端是第一次打开，就也唤醒了正在睡眠等待的进程。

命名管道一经建立，以后的读、写以及关闭操作就与普通管道完全相同了。注意虽然 FIFO 文件的 inode 节点在磁盘上，但那只是一个节点，而文件的数据则只存在于内存缓冲页面中，与普通管道一样。

6.4 信号

如前所述，信号（signal，亦称软中断）机制是在软件层次上对中断机制的一种模拟。从概念上说，一个进程接收到一个信号与一个处理器接收到一个中断请求是一样的。而一个进程可以向另一个（或另一组）进程发送信号，也跟在多处理器系统中一个处理器可以向其它处理器发出中断请求一样。当然，对一个处理器的中断请求并不一定来自其它处理器，也可以是来自各种中断源，甚至来自处理器本身。相应地，信号也不一定都来自其它进程，也可以来自不同的来源，还可以来自本进程的执行。更重要的是，二者都是“异步”的。处理器在执行一段程序时并不需要停下来等待中断的发生，也不知道中断会在何时发生。信号也是一样，一个进程并不需要通过一个什么操作来等待信号的到达，也不知道什么时候会有信号到达。

事实上，在所有的进程间通讯机制中只有信号是异步的。二者之间的这种相似和类比不仅仅是概念上的，也体现在它们的实现上。就像在中断机制中有一个“中断向量表”一样，在每个进程的 task_struct 结构中都有个指针 sig，指向一个 signal_struct 结构，这个结构就不妨称为“信号向量表”。在中断机制中，对每种中断请求都可以加以屏蔽而不让处理器对之作出响应，在信号机制中也有类似的手段。当然，由于中断机制是通过硬件和软件的结合来实现的，而信号则纯粹由软件实现，所以在具体的细节上必然有所不同。但是如果将二者对照起来看，就可以看出信号机制中的有些数据结构和算法实际上就是对中断机制中一些硬件特征的模拟。同时，正是由于二者间的相似，在中断处理中可能碰到的问题和经验一般也适用于信号机制。例如，嵌套中断往往会给程序设计带来一些困难，而嵌套信号也会带来类似的问题。正因为这样，读者在阅读本节时不妨多多回顾和对照“中断与异常”那章中的有关段落。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

人们对信号与中断的相似性（以及其它一些问题）并不是一开始就充分认识和深刻理解的。早期 Unix 系统中的信号机制比较简单和原始，没有充分吸取在中断处理方面所积累的经验，后来在实践中暴露出一些问题而被称为“不可靠信号”。正因为这样，在各种 Unix 的变型版本中就纷纷对信号机制加以扩充，以实现“可靠信号”。在这方面最主要的有 BSD 和 AT&T 分别在 4.2BSD、4.3BSD 和 SVR3 中所作的扩充。但是，这种分别进行的扩充使不同版本间的兼容性成了问题，所以后来又在 POSIX.1 和 POSIX.4 两种标准中对信号机制进行了标准化。其中 POSIX.1 规定了对信号机制的基本要求，而 POSIX.4 则规定了对信号机制的扩充，后者是 POSIX.1 的一个超集。Linux 内核的信号机制符合 POSIX.4 的规定。不过，POSIX 只规定了信号机制的功能和应用界面，并没有规定如何实现。例如，同一种功能可以在操作系统内核中实现，也可以在库程序中实现，所以有些非 Unix 类的操作系统也可能支持 POSIX。

既然信号机制与中断机制在概念上是一致的，我们就从与“中断向量表”相对应的“信号向量表”开始。如前所述，每个进程的 task_struct 结构中都有一个指针 sig，指向一个 signal_struct 结构。这个数据结构类型是在 include/linux/sched.h 中定义的：

==================== include/linux/sched.h 243 247 ====================

```
struct signal_struct {
	atomic_t		count;
	struct k_sigaction	action[_NSIG];
	spinlock_t		siglock;
};
```

结构中的数组 action[]就相当于是一个“信号向量表”，数组中的每个元素就相当于一个“信号向量”，确定了当进程接收到一个具体的信号时应该采取的行动，就好像一个中断向量指向一个中断服务程序一样。不过，“信号向量”有它的特殊之处，除指向一个信号处理程序以外，它还可以是两个特殊常数 SIG_DFL 和 SIG_IGN 之一，分别表示应该对该信号采取“默认”（default）的反应或者忽略而不作任何反应。下面给出这两个常数的定义，见文件 include/asm-i386/signal.h：

==================== include/asm-i386/signal.h 131 133 ====================

```
#define SIG_DFL	((__sighandler_t)0)	/* default signal handling */
#define SIG_IGN	((__sighandler_t)1)	/* ignore signal */
#define SIG_ERR	((__sighandler_t)-1)	/* error return from signal */
```

由于 SIG_DFL 的数值为 0，当向量表为“空白”时所有的信号向量都视作 SIG_DFL。

“信号向量”与“中断向量”还有一个重要的不同之处。大家知道，中断向量表在系统空间中，每个中断向量所指向的中断响应程序也在系统空间中。然而，虽然“信号向量表”也在系统空间中，可是这些“向量”所指向的处理程序却一般都是在用户空间中。

对信号的检测与响应总是发生在系统空间，通常发生在两种情况下：第一，当前进程由于系统调用、中断或异常而进入系统空间以后，从系统空间返回到用户空间的前夕。第二，当前进程在内核中进入睡眠以后刚被唤醒的时候，由于信号的存在而提前返回到用户空间。当有信号要响应时，处理器执行路线的景象如图 6.6 所示。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

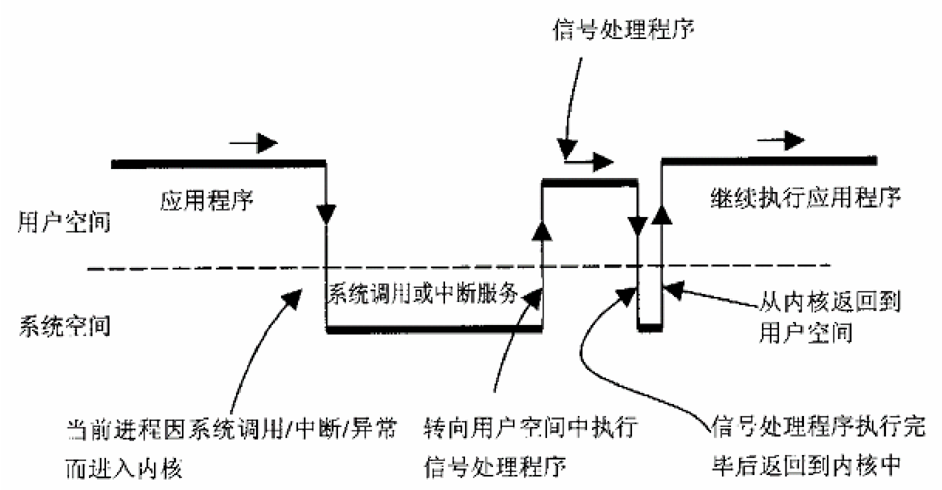


图 6.6　信号的检测与处理流程图

从图中不难看出，信号处理程序（相当于中断服务程序）的启动、执行及返回变得复杂了，读者在后面将会看到这些过程是如何实现的。

中断向量表中的每个“向量”基本上是个函数指针，早期 Unix 系统中的“信号向量表”也是一样。但是，经过扩充与改进，现在的“信号向量”已经不仅仅是函数指针了。每个“信号向量”都是一个 k_sigaction 数据结构，这是在 include/asm-i386/signal.h 中定义的：

==================== include/asm-i386/signal.h 156 164 ====================

```
struct sigaction {
	union {
	  __sighandler_t _sa_handler;
	  void (*_sa_sigaction)(int, struct siginfo *, void *);
	} _u;
	sigset_t sa_mask;
	unsigned long sa_flags;
	void (*sa_restorer)(void);
};
```

这里的 _sa_handler 和 _sa_sigaction 都是函数指针。数据类型 __sighandler_t 也是在 include/asm-i386/signal.h 中定义的：

==================== include/asm-i386/signal.h 129 129 ====================

```
typedef void (*__sighandler_t)(int);
```

可见，_sa_handler 和_sa_sigaction 只是在调用时的参数表不同，具体将_u 解释成哪一个指针取决于具体的约定。

另一个指针 sa_restorer 现在已经基本不用了，但是 sa_mask 和 sa_flags 两个字段却扮演着重要的角色。

先来看 sa_mask。简单地说，sa_mask 是一个“位图”，其中的每一位都对应着一种信号。如果位图中的某一位为 1，就表示在执行当前信号的处理程序期间要将相应的信号暂时“屏蔽”，使得在执行

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的过程中不会嵌套地响应那种信号。特别地，不管位图中的相应位是否为 1，当前信号本身总是自动屏蔽，使得对同一种信号的处理不会嵌套发生，除非 sa_flags 中的 SA_NODEFER 或 SA_NOMASK 标志为 1。显然，这正是借鉴了在中断服务中关闭中断以防止嵌套的经验，对于熟悉中断机制的读者来说似乎并不是什么深奥的道理，可是这却是将“不可靠信号”改进成“可靠信号”的关键性的一步。在早期 Unix 系统的信号机制中，当时的设计人员似乎认为异种信号处理的嵌套不是什么问题，而同种信号处理的嵌套可以通过一种简单的方法来避免。怎么避免呢？那就是：每当执行一个信号处理程序时，就由内核自动将“信号向量表”中相应的函数指针设置成 SIG_DFL。从而，在执行一个信号处理程序的过程中如果又接收到同种信号的话，就会因为此时的“信号向量”已经改成 SIG_DFL 而不会嵌套进入同一个处理程序。这样，应用程序所设置的“信号向量”就是“一次性”的，所以信号处理程序中在完成了需要防止嵌套的部分以后就要再次设置信号向量，为下一次执行同一信号处理程序作好准备。这套方案看起来似乎可行，但是在实践中却碰到了问题。一种典型的情景就是对 CTRL_C 的处理。大家知道，在键盘上启动一个程序后，按一下 CTRL_C 通常会使正在运行的程序“流产”，这实际上就是通过信号机制来实现的。当在键盘上按 CTRL_C 时，内核会向相应的进程发出一个 SIGINT 信号，而对这个信号的“默认”反应就是通过 do_exit()结束运行。有些应用程序对 CTRL_C 的作用另有安排，所以就要为 SIGINT 另行设置一个“向量”，使它指向应用程序中的一个函数，在那个函数中对 CTRL_C 这个事件作出响应，并再次设置（或曰“恢复”）该信号向量，为下一次 CTRL_C 事件作好准备。可是，在实践中却发现，两次 CTRL_C 事件往往过于密集，有时候刚进入信号处理程序，还没有来得及重新设置信号向量，第二个信号就到达了。由于此时向量表中对应于 SIGINT 的向量已在启动其处理程序时自动改变成 SIG_DFL，而对 SIG_DFL 信号的默认反应又是结束进程的运行，所以第二个 SIGINT 信号的到来就往往把进程“杀”了。正因为这样，早期的信号机制被称为“不可靠信号”。从这里人们得出了一些教训。首先信号向量不应该是“一次性”的，也就是不应该在执行相应处理程序时将向量改成 SIG_DFL。其次，在执行一个信号处理程序的过程中应将该种信号自动屏蔽掉，以防同一处理程序的嵌套。此外，还应该有一个手段，使应用程序可以在执行处理程序的期间有选择地将若干种其它信号屏蔽掉，这就是 sa_mask 的来历。所谓“屏蔽”，与将信号忽略丢弃是不同的。它只是将信号暂时“遮盖”一下，一旦屏蔽去除，已经到达的信号仍旧还在。位图 sa_mask 的类型为 sigset_t，这也是在 include/asm-i386/signal.h 中定义的：

==================== include/asm-i386/signal.h 13 21 ====================

```
#define _NSIG      64
#define _NSIG_BPW  32
#define _NSIG_WORDS    (_NSIG / _NSIG_BPW)

typedef unsigned long old_sigset_t;       /* at least 32 bits */

typedef struct {
     unsigned long sig[_NSIG_WORDS];
} sigset_t;
```

这种数据结构主要用来模拟中断控制器（如 Intel 8259）中的“中断请求寄存器”和“中断屏蔽寄存器”，task_struct 结构中的 blocked 就相当于“中断屏蔽寄存器”。以前 task_struct 结构中还有个 sigset_t 数据结构 signal，就是相当于“中断请求寄存器”，后来把它移入了 sigpending 数据结构中（现在 task_struct

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构中有个 sigpending 数据结构 pending)。注意这里的_NSIG 正是前述数组 action[]的大小。早期 Unix 系统中只定义了 32 种信号，所以只要一个无符号长整数就可容纳了。而现在，Linux（以及 POSIX.4）定义了 64 种信号，将来也许还会增加，所以才有以上的定义。

再来看 sa_flags 中的标志位，它们的定义也在 include/asm-i386/signal.h 中：

==================== include/asm-i386/signal.h 73 93 ====================

```
/*
 * SA_FLAGS values:
 *
 * SA_ONSTACK indicates that a registered stack_t will be used.
 * SA_INTERRUPT is a no-op, but left due to historical reasons. Use the
 * SA_RESTART flag to get restarting signals (which were the default long ago)
 * SA_NOCLDSTOP flag to turn off SIGCHLD when children stop.
 * SA_RESETHAND clears the handler when the signal is delivered.
 * SA_NOCLDWAIT flag on SIGCHLD to inhibit zombies.
 * SA_NODEFER prevents the current signal from being masked in the handler.
 *
 * SA_ONESHOT and SA_NOMASK are the historical Linux names for the Single
 * Unix names RESETHAND and NODEFER respectively.
 */
#define SA_NOCLDSTOP    0x00000001
#define SA_NOCLDWAIT    0x00000002 /* not supported yet */
#define SA_SIGINFO 0x00000004
#define SA_ONSTACK 0x08000000
#define SA_RESTART 0x10000000
#define SA_NODEFER 0x40000000
#define SA_RESETHAND    0x80000000
```

这些标志位的作用以后读了有关的代码就会清楚，但是有几个标志位特别值得一提。一个是 SA_SIGINFO。这个标志位为 1 表示信号处理程序有三个参数（否则只有一个，即所处理的信号本身的标识号，如 SIGINT），其中之一为指向一个 siginfo_t 数据结构的指针。与 siginfo_t 有关的数据结构和常数都是在 include/asm-i386/siginfo.h 中定义的：

==================== include/asm-i386/siginfo.h 8 63 ====================

```
typedef union sigval {
    int sival_int;
    void *sival_ptr;
} sigval_t;

#define SI_MAX_SIZE     128
#define SI_PAD_SIZE     ((SI_MAX_SIZE/sizeof(int)) - 3)

typedef struct siginfo {
    int si_signo;
    int si_errno;
    int si_code;

    union {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            int _pad[SI_PAD_SIZE];

            /* kill() */
            struct {
                pid_t _pid;         /* sender's pid */
                uid_t _uid;         /* sender's uid */
            } _kill;

            /* POSIX.1b timers */
            struct {
                unsigned int _timer1;
                unsigned int _timer2;
            } _timer;

            /* POSIX.1b signals */
            struct {
                pid_t _pid;         /* sender's pid */
                uid_t _uid;         /* sender's uid */
                sigval_t _sigval;
            } _rt;

            /* SIGCHLD */
            struct {
                pid_t _pid;         /* which child */
                uid_t _uid;         /* sender's uid */
                int _status;        /* exit code */
                clock_t _utime;
                clock_t _stime;
            } _sigchld;

            /* SIGILL, SIGFPE, SIGSEGV, SIGBUS */
            struct {
                void *_addr; /* faulting insn/memory ref. */
            } _sigfault;

            /* SIGPOLL */
            struct {
                int _band;     /* POLL_IN, POLL_OUT, POLL_MSG */
                int _fd;
            } _sigpoll;
        } _sifields;
    } siginfo_t;
```

如前所述，信号机制是对中断机制的模拟。就像中断请求一样，（早期的）信号所载送的信息是二元的，也就是“有”或“没有”，仅此而已。在中断机制中，一般都是由中断服务程序来读相应外设的状态寄存器以获取进一步的信息，所以传统的信号也得要与其它的手段相结合才能完成一些信息量要求稍大的通信。针对这个缺点，改进后的信号机制使通信的双方可以随信号一起传递一个 siginfo_t 数据结构以及一个 void 指针。其中 siginfo_t 结构的主体是一个 union，根据信号类型 si_signo 的值而赋予

PDF-XCHANGE Click to buy NOW! www.docu-track.com

不同的解释；而 void 指针所指的数据类型则由通信的双方自行约定。

此处先将 Linux 信号的专用名、定义值以及默认反应等项信息列于表 6.1。

表 6.1　Linux 信号专用名、定义值与默认反应

信号名	定义值	用途或来源	默认的反应
SIGHUP	1	控制 TTY 断开连接	进程终止
SIGINT	2	用户在键盘上按 CTRL_C	进程终止
SIGQUIT	3	TTY 键盘上按 CTRL_\	进程流产（core）
SIGILL	4	非法指令（异常）	进程流产（core）
SIGTRAP	5	遇到 debug 断点，用于调试	进程流产（core）
SIGABRT	6	使进程流产	进程流产（core）
SIGIOT	6	同上	同上
SIGBUS	7	访问内存失败	进程流产（core）
SIGFPE	8	算术运算或浮点处理出错	进程流产（core）
SIGKILL	9	使进程终止（不可屏蔽）	进程终止
SIGUSR1	10	由应用软件自行定义和使用	忽略
SIGSEGV	11	越界访问内存	进程流产（core）
SIGUSR2	12	由应用软件自行定义和使用	忽略
SIGPIPE	13	管道断裂（管道的读端已关闭）	进程终止
SIGALRM	14	由 setitimer()设置的定时器到点	忽略
SIGTERM	15	使进程终止	进程终止
SIGSTKFLT	16	用于堆栈出错（尚未使用）	进程终止
SIGCHLD	17	用于子进程终止	忽略
SIGCONT	18	进程继续运行，与 SIGSTOP 结合使用	忽略
SIGSTOP	19	进程暂停运行，转入 TASK_STOPPED 状态	逃程暂停
SIGTSTP	20	CTRL_Z，进程“挂起”（TASK_STOPPED）	进程暂停
SIGTTIN	21	后台进程读控制终端时使用	进程暂停
SIGTTOU	22	后台进程写控制终端时使用	进程暂停
SIGURG	23	用于紧急 I/O 状况	忽略
SIGXCPU	24	进程使用 CPU 已超出限制	进程终止
SIGXFSZ	25	文件大小超出限制	
SIGVTALRM	26	由 setitimer()设置的“虚拟”定时器到点	进程终止
SIGPROF	27	由 setitimer()设置的“统计”定时器到点	进程终止
SIGWINCH	28	控制终端窗口的大小被改变	忽略
SIGIO	29	用于异步 I/O	进程终止
SIGPOLL	29	用于异步 I/o	进程终止
SIGPWR	30	用于电源失效	进程终止
SIGUNUSED	31	保留，未使用	进程终止

PDF-XCHANGE Click to buy NOW! www.docu-track.com

SIGRTMIN	32	从 SIGRTMIN 至 SIGRTMAX 为“实时”信号	进程终止
SIGRTMAX	(_NSIG-1)		

以前，进程的 task_struct 结构中有两个位图（sigset_t 结构）signal 和 blocked（现在 signal 在 task_struct 结构内部的 sigpending 结构 pending 中），分别用来模拟中断控制器硬件中的中断状态（或者说中断请求）寄存器和中断屏蔽寄存器。每当有一个中断请求到来时，中断状态寄存器中与之相应的某一位就被置成 1，表示有相应的中断请求在等待处理，并且一直要到中断响应程序读出这个寄存器时才又被清 0。如果连续有两次中断请求到来，则有可能因为处理器来不及响应而被“合并”，因为中断状态寄存器中具体的状态位一旦被置成 1 以后（在清 0 之前）就反映不出到底被连续几次置 1。在中断机制中这并不是什么问题，因为通常中断响应程序在检测到某个中断“通道”中有中断请求就会“轮询”连接在该通道中所有的中断源，还是可以知道到底有几个中断源发生了请求。所以，在中断机制中这种“合并”效应只是形式上的，不是实质性的。

可是，在信号机制中就不同了。接收到信号的进程无法“轮询”所有可能的信号来源。因此，在信号机制中这种“合并”效应是实质性的。解决这个问题的出路在于为信号准备一个队列，每产生一个信号就把它挂入这个队列，这样就可以确保一个信号也不会丢失了。所以，就在 task_struct 结构中设置了一个信号队列，后来又把 task_struct 结构中的信号队列和信号位图合并成为一个 sigpending 数据结构，定义于 include/linux/signal.h 中：

==================== include/linux/signal.h 17 20 ====================

```
struct sigpending {
      struct sigqueue *head, **tail;
      sigset_t signal;
};
```

顺便提一下，在 task_struct 中还有两个用于信号机制的成分。一个是 sas_ss_sp，用于记录当进程在用户空间执行信号处理程序时的堆栈位置。另一个是 sas_ss_size，那就是堆栈的大小。下面我们列出 task_struct 数据结构中所有与信号有关的成分，以便查阅：

==================== include/linux/sched.h 277 277 ====================

```
struct task_struct {
      . . . . . .
```

==================== include/linux/sched.h 283 283 ====================

```
      int sigpending;
      . . . . . .
```

==================== include/linux/sched.h 317 318 ====================

```
      int exit_code, exit_signal;
      int pdeath_signal;  /*  The signal sent when the parent dies  */
      . . . . . .
```

==================== include/linux/sched.h 379 387 ====================

```
/* signal handlers */
      spinlock_t sigmask_lock;    /* Protects signal and blocked */
      struct signal_struct *sig;

      sigset_t blocked;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	struct sigpending pending;

	unsigned long sas_ss_sp;
	size_t sas_ss_size;
    . . . . . .
```

==================== include/linux/sched.h 397 397 ====================

```
};
```

注意在 task_struct 结构中有个字段叫 sigpending，是个整数，用来表示这个进程是否有信号在等待处理；而上述的 sigpending 数据结构则包括一个信号队列和一个信号位图，不要把两个 sigpending 搞混淆了。

上述的种种改进无疑是很有必要的，但是可惜来迟了。在作出这些改进之前，人们已经在 Unix 上开发了很多使用信号的软件。就信号的使用而言，这些软件在相当程度上可以说是为早期欠成熟的信号机制量身定做的。现在信号机制有了改进，但是对这些已经存在的软件还要保证它们能在新的环境中“发挥余热”，并且不改变其运行时的“习性”。显然，比较简单的办法是保留原先已经定义的（31 个）信号不变（包括有关的系统调用以及使用方式），而另外再定义一些新的信号（和一些新的系统调用），对这些新的信号则实施改进了的信号机制。在实践中，这种对“老编制”实行“老政策”，“新编制”实行“新政策”的现象常常是不可避免的。另一方面，仔细想一下就可以明白，上述的这些改进其实大多是跟时间有关的，所以把新增加的信号称为 RT（实时）信号。从 SIGRTMIN 到 SIGRTMAX 全都是这些 RT 信号。不过，这里要注意不要被 RT 这两个字母迷惑了，这些信号与一般意义上的“实时”并没有关系。例如，这些信号的传递不比普通的信号快，也没有时间上的承诺。

有了上面这些预备知识，我们就可以开始介绍代码了。

先看“信号向量”，也就是信号处理程序的“安装”，其作用类似于中断向量的设置。Linux 为此提供了三个系统调用。第一个是“传统的”、老的调用：

sighandler_t signal(int signum, sighandler_t handler);

这里的数据类型 sighandler_t 为指向信号处理程序的函数指针。调用的参数有两个，一个是信号的定义值 signum（如 SIGINT），另一个就是指向由用户定义的该信号处理程序的函数指针 handler。不过 handler 也可以是两个特殊值之一，即 SIG_IGN 或 SIG_DFL，分别表示忽略该信号或采取对此信号的默认反应。系统调用的返回值为该信号原先的 handler。其它两个系统调用则是新的，但是在用户程序设计界面上却作为相同的库函数出现：

int sigaction (int signum, const struct sigaction *newact, struct sigaction *oldact);

这个库函数会根据信号的编号不同，确定应该落实系统调用 sys_sigaction()还是 sys_rt_sigaction()。

这里的 signum 还是一样，而 newact 和 oldact 为两个指向 sigaction 数据结构的指针。其中 newact 所指向的结构为新的、待设置的向量，而 oldact 所指则为用来返回老向量的数据结构。

像其它系统调用一样，它们在内核中的入口分别为 sys_signal()，sys_sigaction()和 sys_rt_sigaction()。函数 sys_signal()是在 kernel/signal.c 中定义的：

==================== kernel/signal.c 1244 1259 ====================

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * For backwards compatibility.  Functionality superseded by sigaction.
 */
asmlinkage unsigned long
sys_signal(int sig, __sighandler_t handler)
{
	struct k_sigaction new_sa, old_sa;
	int ret;

	new_sa.sa.sa_handler = handler;
	new_sa.sa.sa_flags = SA_ONESHOT | SA_NOMASK;

	ret = do_sigaction(sig, &new_sa, &old_sa);

	return ret ? ret : (unsigned long)old_sa.sa.sa_handler;
}
```

函数 sys_rt_sigaction()也在同一个文件中：

==================== kernel/signal.c 1187 1211 ====================

```
asmlinkage long
sys_rt_sigaction(int sig, const struct sigaction *act, struct sigaction *oact,
		 size_t sigsetsize)
{
	struct k_sigaction new_sa, old_sa;
	int ret = -EINVAL;

	/* XXX: Don't preclude handling different sized sigset_t's.  */
	if (sigsetsize != sizeof(sigset_t))
		goto out;

	if (act) {
		if (copy_from_user(&new_sa.sa, act, sizeof(new_sa.sa)))
			return -EFAULT;
	}

	ret = do_sigaction(sig, act ? &new_sa : NULL, oact ? &old_sa : NULL);

	if (!ret && oact) {
		if (copy_to_user(oact, &old_sa.sa, sizeof(old_sa.sa)))
			return -EFAULT;
	}
out:
	return ret;
}
```

比较一下就可以看出，这两个函数其实都调用 do_sigaction()完成具体的操作。所不同的是，sys_signal()从应用程序得到的信息量较少（只有 handler），同时又要保持早期 Unix 中系统调用 signal()相同的性状，所以将 new_sa.sa.sa_flags 固定设成（SA_ONESHOT | SA_NOMSK）。标志位 SA_ONESHOT 表示所设置的向量是“一次性”的，也就是要按传统的方式，在使用了所设置的函数指针以后就将其

改成 SIG_DFL，而用户空间中的信号处理程序则负有再次调用 signal()恢复该向量的责任。另一个标志位 SA_NOMSK，则表示在执行信号处理程序时不使用任何信号屏蔽，因为最初的信号机制中是没有信号屏蔽这一说的。

另一个函数 sys_sigaction()实际上是与 sys_rt_sigaction()一样的，只是细节略有不同。首先，是在 sys_sigaction()中不存在第四个参数 sigsetsize。其次，也许更重要的，是在 sys_sigaction()参数表中使用的是 old_sigaction 结构指引。，而不是 sigaction 结构指针。这两种数据结构含有相同的成分，但是这些成分在结构中的次序不同，所以在 sys_sigaction()中只好把这些成分逐项地从用户空间复制到系统空间，而不能像在 sys_rt_sigaction()中那样把整个数据结构一次就复制过来。读者也许会好奇，为什么要改变这些成分在数据结构中的次序呢？这不是自找麻烦吗？其实这样做是有道理的。在 old_sigaction 数据结构中，sa_mask 挤在 sa_handler 与 sa_flags 中间，这样就限制了 sa_mask 进一步扩充其大小，也就是定义更多信号的可能。所以，在 sigaction 数据结构中将 sa_mask 移到了末尾，这样当 sa_mask 的大小改变时，虽然 sigaction 的大小也要随之改变，但各个成分在数据结构中的位移却不会改变。考虑到这一点，明知这样做会带来不便也只好“忍痛”了。事实上，对于设计内核代码的人来说，可能带来的混淆还真是不小，因为在应用程序中所用的 sigaction 数据结构却又是对应于内核代码中的 oldsigaction 数据结构！这完全要靠不同的“.h”文件来把它们区分开来。不过，好在内核中用到这个数据结构的代码只是很小一段。

不管怎样，这三个函数最终都是调用 do_sigaction()来完成的，它才是这些系统调用的主体，其代码也在 kernel/signal.c 中（注意，在 arch/i386/kernel 目录下也有个 signal.c）：

==================== kernel/signal.c 1011 1062 ====================

[sys_signal()>do_sigaction()]

```
int
do_sigaction(int sig, const struct k_sigaction *act, struct k_sigaction *oact)
{
	struct k_sigaction *k;

	if (sig < 1 || sig > _NSIG ||
	    (act && (sig == SIGKILL || sig == SIGSTOP)))
		return -EINVAL;

	k = &current->sig->action[sig-1];

	spin_lock(&current->sig->siglock);

	if (oact)
		*oact = *k;

	if (act) {
		*k = *act;
		sigdelsetmask(&k->sa.sa_mask, sigmask(SIGKILL) | sigmask(SIGSTOP));

		/*
		 * POSIX 3.3.1.3:
		 *  "Setting a signal action to SIG_IGN for a signal that is
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1034          *   pending shall cause the pending signal to be discarded,
1035          *   whether or not it is blocked."
1036          *
1037          *  "Setting a signal action to SIG_DFL for a signal that is
1038          *   pending and whose default action is to ignore the signal
1039          *   (for example, SIGCHLD), shall cause the pending signal to
1040          *   be discarded, whether or not it is blocked"
1041          *
1042          * Note the silly behaviour of SIGCHLD: SIG_IGN means that the
1043          * signal isn't actually ignored, but does automatic child
1044          * reaping, while SIG_DFL is explicitly said by POSIX to force
1045          * the signal to be ignored.
1046          */
1047
1048         if (k->sa.sa_handler == SIG_IGN
1049             || (k->sa.sa_handler == SIG_DFL
1050              && (sig == SIGCONT ||
1051                  sig == SIGCHLD ||
1052                  sig == SIGWINCH))) {
1053              spin_lock_irq(&current->sigmask_lock);
1054              if (rm_sig_from_queue(sig, current))
1055                   recalc_sigpending(current);
1056              spin_unlock_irq(&current->sigmask_lock);
1057         }
1058     }
1059
1060     spin_unlock(&current->sig->siglock);
1061     return 0;
1062 }
```

系统对信号 SIGKILL 和 SIGSTOP 的响应是不允许改变的，所以要放在开头时加以检查。同时，这两个信号也不允许被屏蔽，所以要在屏蔽位图 k->sa.sa_mask 中将与这两个信号对应的屏蔽位清除（见 1029 行）。信号的数值是从 1 开始定义的，所以在用信号数值作为数组下标时要用 sig-1 而不是 sig（见 1020 行）。注意 1024 行和 1028 行中的赋值都是整个数据结构的赋值。

当新设置的向量为 SIG_IGN 时，或者为 SIG_DFL 而涉及的信号为 SIGCONT、SIGCHLD 和 SIGWINCH 之一时，如果已经有一个或几个这样的信号在等待处理，则按 POSIX 标准的规定要将这些已经到达的信号丢弃，所以通过 rm_sig_from_queue() 丢弃已经到达的信号。该函数的代码在 kernel/signal.c 中：

==================== kernel/signal.c 294 303 ====================

[sys_signal()>do_sigaction()>rm_sig_from_queue()]

```
/*
 * Remove signal sig from t->pending.
 * Returns 1 if sig was found.
 *
 * All callers must be holding t->sigmask_lock.
 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static int rm_sig_from_queue(int sig, struct task_struct *t)
{
	return rm_from_queue(sig, &t->pending);
}
```

函数 rm_from_queue()也在同一文件中：

==================== kernel/signal.c 270 292 ====================

[sys_signal()>do_sigaction()>rm_sig_from_queue()>rm_from_queue()]

```
270 static int rm_from_queue(int sig, struct sigpending *s)
271 {
272 	struct sigqueue *q, **pp;
273 
274 	if (!sigismember(&s->signal, sig))
275 		return 0;
276 
277 	sigdelset(&s->signal, sig);
278 
279 	pp = &s->head;
280 
281 	while ((q = *pp) != NULL) {
282 		if (q->info.si_signo == sig) {
283 			if ((*pp = q->next) == NULL)
284 				s->tail = pp;
285 			kmem_cache_free(sigqueue_cachep,q);
286 			atomic_dec(&nr_queued_signals);
287 			continue;
288 		}
289 		pp = &q->next;
290 	}
291 	return 1;
292 }
```

对于传统的“老信号”来说，一个进程是否有信号等待着处理，是由 task_struct 结构中的位图 signal 来反映的，位图中的某一位为 1 就表示已经接收到了相应的信号尚未处理，但是却无从知道究竟接收到了几个同种的信号。在这种情况下只是简单地将位图中相应的标志位清 0（见 277 行）。而对于新的“实时”信号，则信号的到达不光是定性的，也是定量的。到达的信号除了使位图中的相应标志位变成 1 以外还进入了本进程的信号队列，所以还要在队列中搜索并将其释放（见 281 行的 while 循环）。

回到 do_sigaction()的代码中，由于丢弃了若干已经到达的信号，当前进程的 task_struct 结构中表示是否有（任何）信号在等待处理的总标志 sigpending 也得要重新算一下了（见 1055 行）。读者也许会问，为什么 task_struct 结构中要有那么个总标志？判断一下 sigpending 结构中的位图 signal 是否为 0 不就可以了吗？问题在于位图并不就是一个长整数。从前，当信号的数量少于 32 个时，那样确实是可以的，但现在不行了。

跟设置“信号向量”有关的系统调用还有一些：

sigprocmask()——改变本进程 task_struct 结构中的信号屏蔽位图 blocked。注意这个屏蔽位图与具体的向量 k_sigaction 数据结构中的屏蔽位图 sa_mask 不同。位图 sa_mask 的屏蔽位只在执行相应的处

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

理程序时才起作用，而 blocked 中的屏蔽位则一直都起作用。还要注意，所谓“屏蔽”的意思只是暂时阻止对已经到达的信号作出响应，一旦屏蔽取消，这些已经到达的信号还是会得到处理的。

sigpending()——检查有哪些信号已经到达而尚未处理。

sigsuspend()——暂时改变本进程的信号屏蔽位图，并使进程进入睡眠，等待任何一个未被屏蔽的信号到达。

这些系统调用也都有相应的“实时信号”版本，如 rt_sigsuspend()等。所有这些系统调用的实现大都在 arch/i386/kernel/signal.c 和 kernel/signal.c 两个文件中。一来限于篇幅，二来也给读者留下举一反三的空间，这里就不深入到有关的代码中去了。

“信号向量”设置好了，就作好了接收和处理信号的准备，下一步就要看怎样向一个进程发送信号了。

同样，发送信号的系统调用也有新旧不同的版本。老的版本是 kill()：

```
int kill(pid_t pid, int sig);
```

参数 pid 为目标进程的 pid，当 pid 为 0 时，表示发送给当前进程所在进程组中所有的进程，为-1 时则发送给系统中的所有进程。

新的版本为 sigqueue()：

```
int sigqueue(pid_t pid, int sig, const union sigval val);
```

与 kill()不同的是，sigqueue()发送的除信号 sig 本身外还有附加的信息，就是 val。此外，sigqueue()只能将信号发送给一个特定的进程，而不像 kill()那样可以通过将参数 pid 设成 0 来发送给整个进程组。参数 val 是一个 union，它可以是一个长整数，但实际上总是一个指向 siginfo 数据结构的指针。

在 clib 中还有个库函数 raise(int sig)，用来发送一个信号给自己，相当于 kill(getpid(), sig)。

系统调用 kill()在内核中的主体为 sys_kill()，其代码在 kernel/signal.c 中：

==================== kernel/signal.c 979 991 ====================

```
asmlinkage long
sys_kill(int pid, int sig)
{
	struct siginfo info;

	info.si_signo = sig;
	info.si_errno = 0;
	info.si_code = SI_USER;
	info.si_pid = current->pid;
	info.si_uid = current->uid;

	return kill_something_info(sig, &info, pid);
}
```

这段代码很简单，先准备下一个 siginfo 结构，然后调用 kill_something_info()：

==================== kernel/signal.c 651 682 ====================

[sys_kill()>kill_something_info()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * kill_something_info() interprets pid in interesting ways just like kill(2).
 *
 * POSIX specifies that kill(-1,sig) is unspecified, but what we have
 * is probably wrong.  Should make it like BSD or SYSV.
 */

static int kill_something_info(int sig, struct siginfo *info, int pid)
{
     if (!pid) {
          return kill_pg_info(sig, info, current->pgrp);
     } else if (pid == -1) {
          int retval = 0, count = 0;
          struct task_struct * p;

          read_lock(&tasklist_lock);
          for_each_task(p) {
               if (p->pid > 1 && p != current) {
                    int err = send_sig_info(sig, info, p);
                    ++count;
                    if (err != -EPERM)
                         retval = err;
               }
          }
          read_unlock(&tasklist_lock);
          return count ? retval : -ESRCH;
     } else if (pid < 0) {
          return kill_pg_info(sig, info, -pid);
     } else {
          return kill_proc_info(sig, info, pid);
     }
}
```

可见，之所以需要 kill_something_info()这一层，是因为要根据 pid 的值来确定是要将信号发送给一个特定的进程（通过 kill_proc_info()），还是整个进程组（通过 kill_pg_info()），还是全部进程？

相比之下，另一个系统调用 sigqueue()只允许将信号发送给一个特定的进程，并且随同信号发送的 siginfo 结构也是由用户进程自己设置的，所以它在内核中的实现 sys_rt_sigqueue()就要简单得多（也在 kernel/signal.c 中）。

==================== kernel/signal.c 993 1009 ====================

```
asmlinkage long
sys_rt_sigqueueinfo(int pid, int sig, siginfo_t *uinfo)
{
     siginfo_t info;

     if (copy_from_user(&info, uinfo, sizeof(siginfo_t)))
          return -EFAULT;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    /* Not even root can pretend to send signals from the kernel.
       Nor can they impersonate a kill(), which adds source info.  */
    if (info.si_code >= 0)
        return -EPERM;
    info.si_signo = sig;

    /* POSIX.1b doesn't mention process groups.  */
    return kill_proc_info(sig, &info, pid);
}
```

这里的 kill_proc_info()根据 pid 找到目标进程的 task_struct 结构，然后通过 send_sig_info()，将信号发送给它（kernel/signal.c）：

==================== kernel/signal.c 635 648 ====================

[sys_kill()>kill_something_info()>kill_proc_info()]

```
inline int
kill_proc_info(int sig, struct siginfo *info, pid_t pid)
{
    int error;
    struct task_struct *p;

    read_lock(&tasklist_lock);
    p = find_task_by_pid(pid);
    error = -ESRCH;
    if (p)
        error = send_sig_info(sig, info, p);
    read_unlock(&tasklist_lock);
    return error;
}
```

而 kill_pg_info()则将同一信号发送给整个进程组：

==================== kernel/signal.c 582 606 ====================

[sys_kill()>kill_something_info()>kill_pg_info()]

```
/*
 * kill_pg_info() sends a signal to a process group: this is what the tty
 * control characters do (^C, ^Z etc)
 */

int
kill_pg_info(int sig, struct siginfo *info, pid_t pgrp)
{
    int retval = -EINVAL;
    if (pgrp > 0) {
        struct task_struct *p;

        retval = -ESRCH;
        read_lock(&tasklist_lock);
        for_each_task(p) {
```

```
                if (p->pgrp == pgrp) {
                        int err = send_sig_info(sig, info, p);
                        if (retval)
                                retval = err;
                }
        }
        read_unlock(&tasklist_lock);
    }
    return retval;
}
```

可见，kill_pg_info()最终也是逐个地找到同一进程组中所有进程的 task_struct 结构，并调用 send_sig_info()将信号发送给它们，也就是说，最后都是通过 send_sig_info()来完成的。我们在第 4 章中讲系统调用 exit()时提到过这个函数，但当时没有深入到它的代码中。现在，让我们来看看到底是怎么发送的。其代码在 kernel/signal.c 中。这个函数比较大，所以还是分段来看（kernel/signal.c）：

==================== kernel/signal.c 503 527 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()]

```
int
send_sig_info(int sig, struct siginfo *info, struct task_struct *t)
{
    unsigned long flags;
    int ret;


#if DEBUG_SIG
printk("SIG queue (%s:%d): %d ", t->comm, t->pid, sig);
#endif

    ret = -EINVAL;
    if (sig < 0 || sig > _NSIG)
        goto out_nolock;
    /* The somewhat baroque permissions check... */
    ret = -EPERM;
    if (bad_signal(sig, info, t))
        goto out_nolock;

    /* The null signal is a permissions and process existance probe.
       No signal is actually delivered.  Same goes for zombies. */
    ret = 0;
    if (!sig || !t->sig)
        goto out_nolock;

```

首先是对输入参数的检查，即所谓“健康检查”，这是通过 bad_signal()进行的（kernel/signal.c）：

==================== kernel/signal.c 305 315 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>bad_signal()]

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Bad permissions for sending the signal
 */
int bad_signal(int sig, struct siginfo *info, struct task_struct *t)
{
     return (!info || ((unsigned long)info != 1 && SI_FROMUSER(info)))
         && ((sig != SIGCONT) || (current->session != t->session))
         && (current->euid ^ t->suid) && (current->euid ^ t->uid)
         && (current->uid ^ t->suid) && (current->uid ^ t->uid)
         && !capable(CAP_KILL);
}
```

这里的 current 指向当前进程（信号的发送者）的 task_struct 结构，t 则指向目标进程（信号的接收者）的 task_struct 结构。宏操作 SI_FROMUSER()以及有关的一些定义都在 include/asm-i386/siginfo.h 中：

==================== include/asm-i386/siginfo.h 99 112 ====================

```
/*
 * si_code values
 * Digital reserves positive values for kernel-generated signals.
 */
#define SI_USER        0         /* sent by kill, sigsend, raise */
#define SI_KERNEL  0x80      /* sent by the kernel from somewhere */
#define SI_QUEUE   -1        /* sent by sigqueue */
#define SI_TIMER __SI_CODE(__SI_TIMER,-2) /* sent by timer expiration */
#define SI_MESGQ   -3        /* sent by real time mesq state change */
#define SI_ASYNCIO -4        /* sent by AIO completion */
#define SI_SIGIO   -5        /* sent by queued SIGIO */

#define SI_FROMUSER(siptr)  ((siptr)->si_code <= 0)
#define SI_FROMKERNEL(siptr)    ((siptr)->si_code > 0)
```

上述的 7 个常数用于 siginfo 结构中的 si_code 字段，用来区分 7 种不同的信号源，读者可以结合看一下前面 sys_kill()中的 986 行。在 sys_rt_sigqueueinfo()中，则随同 siginfo 结构一起来自进程的用户空间，其值必须为一负数。

信号一般只能发送给属于同一个 session（见第 4 章有关说明）以及同一个用户（见“文件系统”一章）的进程，除非发送信号的进程可以通过 suser()暂时性地得到特权用户的权限。代码中的 capable(CAP_KILL)正是试图取得这种特权，读者可参阅“文件系统”一章中的有关内容。

这里要提醒一下。在上面的 if 语句中，capable(CAP_KILL)出现在一个“与”条件表达式中，所以只有在前面的所有各项均为 true 时才会执行，这是由 C 语言的语义规则决定的。还有，这里的异或运算，如(current->euid ^ t->suid)，实际上就是检验两者是否不等。

我们假定通过了所有这些检验，继续往下看（kernel/signal.c）：

==================== kernel/signal.c 528 536 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()]

```
     spin_lock_irqsave(&t->sigmask_lock, flags);
     handle_stop_signal(sig, t);
```

```

    /* Optimize away the signal, if it's a signal that can be
       handled immediately (ie non-blocked and untraced) and
       that is ignored (either explicitly or by default).  */

    if (ignored_signal(sig, t))
        goto out;
```

前面讲过，一个进程可以通过信号屏蔽位图来暂时扣压（或遮盖）所接收到的信号。但是，在接收到某些特定的信号以后，就不容许屏蔽另一些特定的后续信号，所以对这些信号要强行除去屏蔽，为后续信号的处理扫清道路。例如，在接收到 SIGSTOP 以后，其后续信号必然是 SIGCONT，所以要将屏蔽位图中的 SIGCONT 屏蔽位强行清 0。而 SIGCONT 的后续信号则可以是 SIGSTOP、SIGTSTP、SIGTTOU 以及 SIGTTIN 中的任何一个，所以要把这些屏蔽位全部清除。另一方面，对于 SIGKILL 和 SIGCONT 来说，如果目标进程正在 TASK_STOPPED 状态（注意，不是睡眠状态），还要将其唤醒，也就是将进程的状态改成 TASK_RUNNING。这些处理都是由 handle_stop_signal()完成的。其代码也在同一文件中：

==================== kernel/signal.c 374 400 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>handle_stop_signal()]

```
/*
 * Handle TASK_STOPPED cases etc implicit behaviour
 * of certain magical signals.
 *
 * SIGKILL gets spread out to every thread.
 */
static void handle_stop_signal(int sig, struct task_struct *t)
{
    switch (sig) {
    case SIGKILL: case SIGCONT:
        /* Wake up the process if stopped.  */
        if (t->state == TASK_STOPPED)
            wake_up_process(t);
        t->exit_code = 0;
        rm_sig_from_queue(SIGSTOP, t);
        rm_sig_from_queue(SIGTSTP, t);
        rm_sig_from_queue(SIGTTOU, t);
        rm_sig_from_queue(SIGTTIN, t);
        break;

    case SIGSTOP: case SIGTSTP:
    case SIGTTIN: case SIGTTOU:
        /* If we're stopping again, cancel SIGCONT */
        rm_sig_from_queue(SIGCONT, t);
        break;
    }
}
```

回到 send_sig_info()的代码中，535 行是一项优化。如果目标进程的“信号向量表”中对所投递信号的响应是“忽略”（SIG_IGN），并且不在跟踪模式中，也没有加以屏蔽，那就根本不用投递了（除 SIGCHLD 外）。函数 ignored_signal()的代码在 kernel/signal.c 中：

==================== kernel/signal.c 365 372 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>ignored_signal()]

```
static int ignored_signal(int sig, struct task_struct *t)
{
	/* Don't ignore traced or blocked signals */
	if ((t->ptrace & PT_PTRACED) || sigismember(&t->blocked, sig))
		return 0;

	return signal_type(sig, t->sig) == 0;
}
```

==================== kernel/signal.c 317 354 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>ignored_signal()>signal_type()]

```
/*
 * Signal type:
 *    < 0 : global action (kill - spread to all non-blocked threads)
 *    = 0 : ignored
 *    > 0 : wake up.
 */
static int signal_type(int sig, struct signal_struct *signals)
{
	unsigned long handler;

	if (!signals)
		return 0;

	handler = (unsigned long) signals->action[sig-1].sa.sa_handler;
	if (handler > 1)
		return 1;

	/* "Ignore" handler.. Illogical, but that has an implicit handler for SIGCHLD */
	if (handler == 1)
		return sig == SIGCHLD;

	/* Default handler. Normally lethal, but.. */
	switch (sig) {

	/* Ignored */
	case SIGCONT: case SIGWINCH:
	case SIGCHLD: case SIGURG:
		return 0;

	/* Implicit behaviour */
	case SIGTSTP: case SIGTTIN: case SIGTTOU:
		return 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    /* Implicit actions (kill or do special stuff) */
    default:
        return -1;
    }
}
```

不然的话，就进入具体投递的过程了。我们继续往下看（kernel/signal.c）：

==================== kernel/signal.c 538 555 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()]

```
538     /* Support queueing exactly one non-rt signal, so that we
539        can get more detailed information about the cause of
540        the signal. */
541     if (sig < SIGRTMIN && sigismember(&t->pending.signal, sig))
542         goto out;
543 
544     ret = deliver_signal(sig, info, t);
545 out:
546     spin_unlock_irqrestore(&t->sigmask_lock, flags);
547     if ((t->state & TASK_INTERRUPTIBLE) && signal_pending(t))
548         wake_up_process(t);
549 out_nolock:
550 #if DEBUG_SIG
551 printk(" %d -> %d\n", signal_pending(t), ret);
552 #endif
553 
554     return ret;
555 }
```

对于“老编制”的信号（sig < SIGRTMIN），所谓“投递”本来是很简单的，因为那只是将目标进程的“接收信号位图 ” signal 中相应的标志位设置成 1，而无需将信号挂入队列。以前讲过，这样的机制有时候会将在短时期中接收到的多个同种信号合并成一个。但是，内核中对这些“老编制”信号也套用了 siginfo 数据结构（见 sys_kill()的代码），尽管这个数据结构中的信息并非来自应用程序，也并不完整，但多少总也载送着一些信息。所以，这里采用了一种折中的办法，就是对于“老编制”的信号也将其 siginfo 结构挂入队列中，不过只挂入一次。以 SIGINT 为例，当第一个 SIGINT 到达时，接收位图中的 SIGINT 标志位为 0，所以将其设置成 1，并且将伴随的 siginfo 结构挂入队列。然后，如果在第一个 SIGINT 信号尚未处理时第二个 SIGINT 又到来了，则此时接收位图中相应的标志位已经为 1，队列中已经有一个 SIGINT 的 siginfo 数据结构在等待处理，所以就不需要再投递了。这就是 541 行中 sigismember()所作的测试。所以，在来不及处理的情况下，相继到达的同种信号就合并了。这样的实现一来是多少也增加了一些信息量，二来读者以后会看到简化了对信号作出响应时的代码。同时，这样的实现也与传统的信号机制在语义上完全一致。

如果到达的信号属于“新编制”，即“实时信号”，或者虽属“老编制”，但接收位图中相对的标志位为 0，那就要通过 deliver_signal()“投递”信号了（kernel/signal.c）。

==================== kernel/signal.c 493 501 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>deliver_signal()]

```
static int deliver_signal(int sig, struct siginfo *info, struct task_struct *t)
{
	int retval = send_signal(sig, info, &t->pending);

	if (!retval && !sigismember(&t->blocked, sig))
		signal_wake_up(t);

	return retval;
}
```

具体的操作主要是 send_signal()，其代码也在 kernel/signal.c 中：

==================== kernel/signal.c 402 453 ====================

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>deliver_signal()>send_signal()]

```
static int send_signal(int sig, struct siginfo *info, struct sigpending *signals)
{
	struct sigqueue * q = NULL;

	/* Real-time signals must be queued if sent by sigqueue, or
	   some other real-time mechanism.  It is implementation
	   defined whether kill() does so.  We attempt to do so, on
	   the principle of least surprise, but since kill is not
	   allowed to fail with EAGAIN when low on memory we just
	   make sure at least one signal gets delivered and don't
	   pass on the info struct.  */

	if (atomic_read(&nr_queued_signals) < max_queued_signals) {
		q = kmem_cache_alloc(sigqueue_cachep, GFP_ATOMIC);
	}

	if (q) {
		atomic_inc(&nr_queued_signals);
		q->next = NULL;
		*signals->tail = q;
		signals->tail = &q->next;
		switch ((unsigned long) info) {
			case 0:
				q->info.si_signo = sig;
				q->info.si_errno = 0;
				q->info.si_code = SI_USER;
				q->info.si_pid = current->pid;
				q->info.si_uid = current->uid;
				break;
			case 1:
				q->info.si_signo = sig;
				q->info.si_errno = 0;
				q->info.si_code = SI_KERNEL;
				q->info.si_pid = 0;
```

```
436                 q->info.si_uid = 0;
437                 break;
438             default:
439                 copy_siginfo(&q->info, info);
440                 break;
441         }
442     } else if (sig >= SIGRTMIN && info && (unsigned long)info != 1
443            && info->si_code != SI_USER) {
444         /*
445          * Queue overflow, abort.  We may abort if the signal was rt
446          * and sent by user using something other than kill().
447          */
448         return -EAGAIN;
449     }
450 
451     sigaddset(&signals->signal, sig);
452     return 0;
453 }
```

投递时将 siginfo 结构的内容复制到一个 sigqueue 数据结构中，并将这个结构挂入队列。sigqueue 数据结构的定义在 include/linux/signal.h 中：

==================== include/linux/signal.h 12 15 ====================

```
12  struct sigqueue {
13      struct sigqueue *next;
14      siginfo_t info;
15  };
```

读者也许会问，为什么不直接在 siginfo_t 数据结构中增加一个指针 next，从而直接将此数据结构挂入队列中呢？这是因为并非每次调用 sys_kill()或 sys_rt_sigqueueinfo()时都一定会将这个数据结构挂入队列，而分配 / 释放这样一个数据结构的系统开销实际上远超过当真有必要时临时加以复制的开销。所以，在 sys_kill()和 sys_rt_sigqueueinfo()中，将这个 siginfo_t 结构作为局部量安排在堆栈上，只在确有必要时才分配一个 signal_queue 数据结构并加以复制。

不过，并非在所有的情况下都是将伴随着（或者说载送着）信号的 siginfo_t 结构复制到 sigqueue 结构中，有两种情况是例外的，那就是当参数 info 的值为特殊值 0 或 1 的时候。在这两种情况下，info 应被视作整数而不是指针，发生的信号来自系统空间（而不是由一个进程在用户空间中通过系统调用发出）的情况下。此时由 send_sig_info()补充产生出相应的内容（见 424～437 行）。

最后，还要通过 sigaddset()将接收位图中相应的标志位设置成 1。

成功地投递了信号并不说明这个信号就是在等待着处理了，还要看目标进程是否屏蔽了这个信号，所以回到 deliver_signal()中要调用 sigismember()加以检查（497 行）。如果目标进程正在睡眠中，并且没有屏蔽所投递的信号，就要将其唤醒并立即进行调度（498 行）。

函数 signal_wake_up()的代码在 kernel/signal.c 中。我们把有关的代码列出在这里，读者可结合“进程调度”自行阅读：

==================== kernel/signal.c 455 491 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_kill()>kill_something_info()>kill_proc_info()>send_sig_info()>deliver_signal()>signal_wake_up()]

```
/*
 * Tell a process that it has a new active signal..
 *
 * NOTE! we rely on the previous spin_lock to
 * lock interrupts for us! We can only be called with
 * "sigmask_lock" held, and the local interrupt must
 * have been disabled when that got acquired!
 *
 * No need to set need_resched since signal event passing
 * goes through ->blocked
 */
static inline void signal_wake_up(struct task_struct *t)
{
     t->sigpending = 1;

     if (t->state & TASK_INTERRUPTIBLE) {
          wake_up_process(t);
          return;
     }

#ifdef CONFIG_SMP
     /*
      * If the task is running on a different CPU
      * force a reschedule on the other CPU to make
      * it notice the new signal quickly.
      *
      * The code below is a tad loose and might occasionally
      * kick the wrong CPU if we catch the process in the
      * process of changing - but no harm is done by that
      * other than doing an extra (lightweight) IPI interrupt.
      */
     spin_lock(&runqueue_lock);
     if (t->has_cpu && t->processor != smp_processor_id())
          smp_send_reschedule(t->processor);
     spin_unlock(&runqueue_lock);
#endif /* CONFIG_SMP */
}
```

函数 wake_up_process()的代码在第 4 章中已经读过，读者不妨重温一下。这里只是提一下：将目标进程唤醒以后，如果目标进程的优先级别高于当前进程，那么在当前进程从系统调用返回之际就有可能进行一次调度，而目标进程是否能被调度运行，则取决于其优先级别及其它各种因素。下面还要谈这个问题。

并非所有的信号都是由某个进程在用户空间通过系统调用发送的。例如，在第 2 章中的页面异常处理程序 do_page_fault()里，当页面异常无法恢复时就会通过 force_sig()向当前进程发送一个 SIGBUS 信号。函数 force_sig()是内核中发送信号的基本手段。其代码在 kernel/signal.c 中：

==================== kernel/signal.c 694 698 ====================

[do_page_fault()>force_sig()]

```
void
force_sig(int sig, struct task_struct *p)
{
	force_sig_info(sig, (void*)1L, p);
```

==================== kernel/signal.c 562 580 ====================

[do_page_fault()>force_sig()>force_sig_info()]

```
int
force_sig_info(int sig, struct siginfo *info, struct task_struct *t)
{
	unsigned long int flags;

	spin_lock_irqsave(&t->sigmask_lock, flags);
	if (t->sig == NULL) {
		spin_unlock_irqrestore(&t->sigmask_lock, flags);
		return -ESRCH;
	}

	if (t->sig->action[sig-1].sa.sa_handler == SIG_IGN)
		t->sig->action[sig-1].sa.sa_handler = SIG_DFL;
	sigdelset(&t->blocked, sig);
	recalc_sigpending(t);
	spin_unlock_irqrestore(&t->sigmask_lock, flags);

	return send_sig_info(sig, info, t);
}
```

注意，在 force_sig()中调用 force_sig_info()时将第二个参数设成 1，表示是以内核的名义发送的。正因为 force_sig()是“强制”目标进程接收一个信号，所以不允许目标进程忽略该信号，并且在调用 send_sig_info()前要将其屏蔽位也强制清除。

至此，信号的投递已经完成，接下来就是目标进程如何发现信号的到来以及如何对此作出反应了。

在中断机制中，处理器的硬件在每条指令结束时都要检测是否有中断请求存在。信号机制是纯软件的，当然不能依靠硬件来检测信号的到来。同时，要在每条指令结束时都来检测显然也是不现实的、甚至是不可能的。那么，一个进程在什么情况下检测信号的存在呢？首先是每当从系统调用、中断处理或异常处理返回到用户空间的前夕；还有就是当进程被从睡眠中唤醒（必定是在系统调用中）的时候，此时若发现有信号在等待就要提前从系统调用返回。总而言之，不管是正常返回还是提前返回，在返回到用户空间的前夕总是要检测信号的存在并作出反应，这一点我们在第 3 章中已经提到过。下面是 arch/i386/kernel/entry.S 中的一个片断：

==================== arch/i386/kernel/entry.S 217 233 ====================

```
ret_with_reschedule:
	cmpl $0,need_resched(%ebx)
	jne reschedule
	cmpl $0,sigpending(%ebx)
	jne signal_return
restore_all:
```

```
	RESTORE_ALL

	ALIGN
signal_return:
	sti				# we can get here from an interrupt handler
	testl $(VM_MASK),EFLAGS(%esp)
	movl %esp,%eax
	jne v86_signal_return
	xorl %edx,%edx
	call SYMBOL_NAME(do_signal)
	jmp restore_all
```

建议读者回过头去看一下第 3 章中的有关内容，搞清楚 sigpending(%ebx)就是 current->sigpending。这里还要指出一点，一般来说，当进程运行于用户空间时，即使信号到达了也不会引起进程立刻对信号作出反应，而要到当前进程因某种原因（包括时钟中断）进入内核并从内核返回时才会作出反应，所以通常在时间上都会有一段延迟。可是，当信号来源于异常处理（或中断服务、系统调用）时，则由于进程已经在内核中运行，在返回到用户空间之前就会作出反应，所以几乎可以认为是立即就作出反应。特别是在异常处理时，这种反应发生在返回到用户空间重新执行引起异常的那条指令之前，所以从用户空间的程序执行角度来看就是立即的。

对信号作出反应的具体操作是通过 do_signal()完成的。这又是一个比较大的函数，我们还是一段一段往下看。有关的代码都在 arch/i386/kernel/signal.c 中：

==================== arch/i386/kernel/signal.c 579 600 ====================

[ret_with_reschedule()>do_signal()]

```
/*
 * Note that 'init' is a special process: it doesn't get signals it doesn't
 * want to handle. Thus you cannot kill init even with a SIGKILL even by
 * mistake.
 */
int do_signal(struct pt_regs *regs, sigset_t *oldset)
{
	siginfo_t info;
	struct k_sigaction *ka;

	/*
	 * We want the common case to go fast, which
	 * is why we may in certain cases get here from
	 * kernel mode. Just return without doing anything
	 * if so.
	 */
	if ((regs->xcs & 3) != 3)
		return 1;

	if (!oldset)
		oldset = &current->blocked;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

读者不妨先回顾一下第 3 章中有关 pt_regs 数据结构的内容。在中断处理、异常处理或系统调用中，处理器的系统空间堆栈保存着该处理器在进入内核之前的“现场”，也就是各个寄存器在进入内核前的内容。Linux 的内核将系统空间堆栈中这些寄存器的“映象”看作一个数据结构，这就是 pt_regs。所以，这里的指针 regs 指向系统空间堆栈中的这些寄存器映象，其中 regs->xcs 为处理器进入这些程序前代码段寄存器 CS 的内容。如果处理器是从用户空间进入中断、异常或陷阱（系统调用），则当时 CS 的最低两位必定是 3（表示用户空间）。反过来，如果 regs->xcs 的最低两位不等于 3 的话，就说明本次中断或异常发生于系统空间，所以处理器并不是处于返回到用户空间的前夕，并不需要对信号作出反应。否则，就要往下跑了（arch/i386/kernel/signal.c：do_signal()）：

==================== arch/i386/kernel/signal.c 601 704 ====================

[ret_with_reschedule()>do_signal()]

```
	for (;;) {
		unsigned long signr;

		spin_lock_irq(&current->sigmask_lock);
		signr = dequeue_signal(&current->blocked, &info);
		spin_unlock_irq(&current->sigmask_lock);

		if (!signr)
			break;

		if ((current->ptrace & PT_PTRACED) && signr != SIGKILL) {
			/* Let the debugger run.  */
			current->exit_code = signr;
			current->state = TASK_STOPPED;
			notify_parent(current, SIGCHLD);
			schedule();

			/* We're back.  Did the debugger cancel the sig?  */
			if (!(signr = current->exit_code))
				continue;
			current->exit_code = 0;

			/* The debugger continued.  Ignore SIGSTOP.  */
			if (signr == SIGSTOP)
				continue;

			/* Update the siginfo structure.  Is this good?  */
			if (signr != info.si_signo) {
				info.si_signo = signr;
				info.si_errno = 0;
				info.si_code = SI_USER;
				info.si_pid = current->p_pptr->pid;
				info.si_uid = current->p_pptr->uid;
			}

			/* If the (new) signal is now blocked, requeue it.  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
		if (sigismember(&current->blocked, signr)) {
			send_sig_info(signr, &info, current);
			continue;
		}
	}

	ka = &current->sig->action[signr-1];
	if (ka->sa.sa_handler == SIG_IGN) {
		if (signr != SIGCHLD)
			continue;
		/* Check for SIGCHLD: it's special.  */
		while (sys_wait4(-1, NULL, WNOHANG, NULL) > 0)
			/* nothing */;
		continue;
	}

	if (ka->sa.sa_handler == SIG_DFL) {
		int exit_code = signr;

		/* Init gets no signals it doesn't want.  */
		if (current->pid == 1)
			continue;

		switch (signr) {
		case SIGCONT: case SIGCHLD: case SIGWINCH:
			continue;

		case SIGTSTP: case SIGTTIN: case SIGTTOU:
			if (is_orphaned_pgrp(current->pgrp))
				continue;
			/* FALLTHRU */

		case SIGSTOP:
			current->state = TASK_STOPPED;
			current->exit_code = signr;
			if (!(current->p_pptr->sig->action[SIGCHLD-1].sa.sa_flags & SA_NOCLDSTOP))
				notify_parent(current, SIGCHLD);
			schedule();
			continue;

		case SIGQUIT: case SIGILL: case SIGTRAP:
		case SIGABRT: case SIGFPE: case SIGSEGV:
		case SIGBUS: case SIGSYS: case SIGXCPU: case SIGXFSZ:
			if (do_coredump(signr, regs))
				exit_code |= 0x80;
			/* FALLTHRU */

		default:
			sigaddset(&current->pending.signal, signr);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
686                         recalc_sigpending(current);
687                         current->flags |= PF_SIGNALED;
688                         do_exit(exit_code);
689                         /* NOTREACHED */
690                     }
691             }
692
693             /* Reenable any watchpoints before delivering the
694              * signal to user space. The processor register will
695              * have been cleared if the watchpoint triggered
696              * inside the kernel.
697              */
698             __asm__("movl %0,%%db7": : "r" (current->thread.debugreg[7]));
699
700             /* Whee!  Actually deliver the signal.  */
701             handle_signal(signr, ka, &info, oldset, regs);
702             return 1;
703         }
704
```

这是一个比较大的 for 循环（601～703 行）。在每一轮循环中都要从进程的信号队列中通过 dequeue_signal()取出一个未加屏蔽的信号加以处理，直到信号队列中不再存在这样的信号（见 608 行），或者相应的“信号向量”为 SIG_DFL，即对该信号采取默认的反应方式为止。而这默认的反应又是让接收到信号的进程“寿终正寝”（见 688 行），或者执行了一个由用户设置的信号处理程序之后（见 702 行）。

函数 dequeue_signal()的代码也在同一文件中，其代码并不复杂却颇为繁琐，我们就不深入进去了。简单地说，它参照一个屏蔽位图，在这里是 current->blocked，从当前进程的信号队列中找到一个未加屏蔽的信号，即 signal_queue 数据结构，将其脱链并把其内容复制到数据结构 info 中，释放该 signal_queue 数据结构，然后将进程的信号接收位图中相应的标志位清 0，并重新计算一下进程的 sigpending 标志。

当一个进程受到其父进程的跟踪而处于 debug 模式时，对信号的反应有一些特殊的考虑（611~641 行）。我们这里先跳过它，到讲述 ptrace()时再回过头来看这一段代码。

如前所述，对具体信号的反应取决于其“信号向量”的设置，所以要根据信号的数值在“信号向量表”中找到相应的向量，即 k_sigaction 数据结构，并让指针 ka 指向这个数据结构。

如果设置的反应方式是“忽略”（SIG_IGN），那么一般来说对这个信号的处理就完成了（见 646 行）。但有一个例外，那就是当信号为 SIGCHLD 时。这个信号通常是在一个子进程通过 exit()系统调用结束其生命时向父进程发出的，所以此时接收到 SIGCHLD 信号的进程要调用 sys_wait4()来检查其所有的子进程（第一个参数为-1），只要找到一个已经结束生命的子进程就为其“料理后事”（见第 4 章）。注意，这里的第三个参数 WNOHANG，表示如果没有发现已经结束生命的子进程也要立即返回（正常返回值为 0），而不作等待。

另一个特殊的反应方式是 SIG_DFL。当“信号向量”为 SIG_DFL 时，多数信号（包括 SIG_KILL）都会落入 684 行的 default 部分，通过 do_exit()结束进程的运行。对于 SIG_KILL 来说，一来它的信号

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

向量是不容许设置的；二来向量中的相应屏蔽位也在每次设置“信号向量”时自动清 0（均见 do_sigaction()），而且在通过 sys_rt_sigprocmask()设置进程的信号屏蔽位图时也会自动将 SIGKILL 的屏蔽位自动清 0。所以对于目标进程，这个信号是头号“杀手”。注意代码中的 667 和 682 行，这些地方都没有 break 语句。不过，pid 为 1 的 init 进程对所有这些信号都有“免疫力”而不受任何影响（见 658 行）。

由此可见，当信号向量为 SIG_IGN 或 SIG_DFL 时，对信号的反应都在系统空间完成，而无须回到用户空间。

如果“信号向量”指向某个由用户设置的信号处理程序，那就要调用 handle_signal()予以执行了。我们将在后面加以介绍。

当 601 行的 for 循环“正常”结束时，也就是说当执行到第 703 行的后面时，当前进程中肯定已经没有未加屏蔽的信号了。这是因为在这个 fo 循环中只有一个出口，即 break 语句，那就是在第 609 行处。而且已经处理过的信号肯定全都不是通过用户定义的信号处理程序进行的，否则就在 702 行处返回了。再进一步，这些信号中也没有一个使进程结束运行，否则就在 688 行通过不会返回的 do_exit()退出了。换言之，这些信号只可能是 SIGCHLD、SIGCONT、SIGWINCH、SIGTSTP、SIGTTIN、SIGTTOU 和 SIGSTOP，或者信号向量设置成 SIG_IGN 的其它信号。这些信号如果来自某个系统调用的过程中，则往往标志着该系统调用过程的失败。这种情况常常发生在设备驱动程序中，并且往往会要求自动重新执行失败的系统调用，就好像在执行指令的过程中发生异常时要求重新执行失败的指令一样。那么，这是怎么实现的呢？让我们继续往下看 do_signal()：

==================== arch/i386/kernel/signal.c 705 716 ====================

[ret_with_reschedule()>do_signat()]

```
	/* Did we come from a system call? */
	if (regs->orig_eax >= 0) {
		/* Restart the system call - no handlers present */
		if (regs->eax == -ERESTARTNOHAND ||
		    regs->eax == -ERESTARTSYS ||
		    regs->eax == -ERESTARTNOINTR) {
		      regs->eax = regs->orig_eax;
		      regs->eip -= 2;
		}
	}
	return 0;
}
```

首先要明白，regs->orig_eax 为处理器进入系统空间前寄存器 EAX 的内容，而 regs->eax 则为系统调用的返回值。处理器在因系统调用而进入系统空间之前，寄存器 EAX 中为系统调用号。而系统调用号不会是负数，所以首先要检查 regs->orig_eax 是否大于等于 0。另一方面，失败的系统调用若要求自动重新执行就会将 EAX 中的返回值 regs->eax 设置成负的 ERESTARTNOHAND，ERESTARTSYS 和 ERESTARTNOINTR 之一。所以，当 regs->eax 为这三者之一时，就将 regs->orig_eax 写回 regs->eax，并且将 regs->eip 的数值减 2。我们在第 3 章中讲过，系统调用是通过一条“int $0x80”指令实现的。在正常的情况下，当处理器执行该指令进入系统空间时其指令指针 EIP 指向其下一条指令，这样当处理器返回到用户空间时就会继续往下执行。现在，将 regs->eip 的数值减 2，就使得处理器返回到用户

空间时其 EIP 又回过去指向该 INT 指令了（因为 INT 指令的大小是两个字节），所以就会重新执行一次该系统调用。

如果用户设置了信号处理程序（在用户空间中），就要通过函数 handle_signal()准备好对处理程序的执行，其代码也在 arch/i386/kernel/signal.c 中：

==================== arch/i386/kernel/signal.c 533 577 ====================

[ret_with_reschedule()>do_signal()>handle_signal()]

```
/*
 * OK, we're invoking a handler
 */

static void
handle_signal(unsigned long sig, struct k_sigaction *ka,
          siginfo_t *info, sigset_t *oldset, struct pt_regs * regs)
{
     /* Are we from a system call? */
     if (regs->orig_eax >= 0) {
          /* If so, check system call restarting.. */
          switch (regs->eax) {
               case -ERESTARTNOHAND:
                    regs->eax = -EINTR;
                    break;

               case -ERESTARTSYS:
                    if (!(ka->sa.sa_flags & SA_RESTART)) {
                         regs->eax = -EINTR;
                         break;
                    }
               /* fallthrough */
               case -ERESTARTNOINTR:
                    regs->eax = regs->orig_eax;
                    regs->eip -= 2;
          }
     }

     /* Set up the stack frame */
     if (ka->sa.sa_flags & SA_SIGINFO)
          setup_rt_frame(sig, ka, info, oldset, regs);
     else
          setup_frame(sig, ka, oldset, regs);

     if (ka->sa.sa_flags & SA_ONESHOT)
          ka->sa.sa_handler = SIG_DFL;

     if (!(ka->sa.sa_flags & SA_NODEFER)) {
          spin_lock_irq(&current->sigmask_lock);
          sigorsets(&current->blocked,&current->blocked,&ka->sa.sa_mask);
          sigaddset(&current->blocked,sig);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            recalc_sigpending(current);
            spin_unlock_irq(&current->sigmask_lock);
        }
}
```

由于在 do_signal()中执行完 handle_signal()以后接着就返回了，所以这里也要考虑系统调用失败后重新执行的问题。不过，此时（执行用户设置的信号处理程序前夕）的重新执行（实际上是为重新执行所作的准备）却是有区分并且有条件的了。

前面讲过，由用户提供的信号处理程序是在用户空间执行的，而且执行完毕以后还要回到系统空间，这是由 setup_rt_frame()或 setup_frame()作出安排的。二者的代码大同小异，所以我们在这里只看 setup_frame()。在深入到代码中去之前，先大致介绍一下所涉及的一些问题和解决方案。大家知道，在调用一个子程序时，堆栈要往下（逻辑意义上是往上）伸展，这是因为需要在堆栈中保存子程序的返回地址，还因为子程序往往会有局部变量，也要占用堆栈中的空间。此外，调用子程序时的参数也是在堆栈中。子程序调用嵌套愈深，则堆栈伸展的层次也愈多。而堆栈中的每一个这样的层次，就称为一个“框架”，即 frame。当子程序和调用它的程序都在同一空间中时，堆栈的伸展，也就是堆栈中框架的建立是很自然的。因为首先 call 指令本身就会将返回地址自动压入堆栈，而调用参数则通过 push 指令压入堆栈。其次，在堆栈中为局部变量分配空间也很简单，只要在进入子程序之后适当调整堆栈指针就可以了。然而，当二者不在同一空间时，情况就比较复杂了。从某种意义上讲，中断处理、异常处理以及系统调用，都可以看作是子程序调用，只不过调用者在用户空间而子程序在系统空间。所以，在返回到用户空间前夕，系统空间堆栈的内容，也就是指针 regs 所指向的 pt_regs 数据结构，实际上就是一个框架。这个框架的内容决定了当处理器回到用户空间时从何处继续执行指令，用户空间堆栈在何处以及各个寄存器的内容。现在，既然要求处理器在回到用户空间时要执行另一段程序，就得在系统空间堆栈中为之准备一个不同的框架。可是，最终还得要回到当初作出系统调用或者被中断的地方去，所以原先的框架也不能丢掉，要保存起来。保存在那里呢？一个进程的系统空间堆栈的大小是很有限的（见第 3 章），所以最合理的就是把它作为信号处理程序的附加局部量，也就是保存在进程的用户空间堆栈中的因调用该处理程序而形成的框架里。这样，就有必要在进入用户空间执行信号处理程序之前，就准备好用户空间堆栈中的框架，只有如此才能先把原先的框架复制到用户空间的框架中作为局部量保存起来，回到系统空间中以后再从那里复制回来。框架的形成是在程序运行过程中，特别是在子程序调用的过程中自然形成的，但是框架的形成也有其规律可循。现在尚未执行对信号处理程序的调用，当然也不存在调用该处理程序的框架，所以实际上是按照形成框架的规律先作好准备，预先在用户空间堆栈中打下一些埋伏。

另一个问题是，在用户空间执行完信号处理程序以后，又怎样重返系统空间？我们知道，从用户空间进入系统空间的手段无非就是中断、异常以及陷阱，而系统调用正是陷阱的运用。显然，中断和异常都不如系统调用更为合适，所以内核中为了这个目的而专门设置了一个系统调用 sigreturn()。不过，要求用户在其信号处理程序中调用一个特别的库函数或系统调用是不大合适的。因为一来那样就对信号处理程序有了特殊的要求，二来更难以保证用户不会忘记在其程序中作出这样的调用，并且 C 编译也难以保证在编译时加以检查（当然，也并非绝对不可能）。所以，最好是由内核在启动信号处理程序时自动地插入这个调用。

这样，思路就渐渐清晰了。整个过程大致上可以归纳为以下这些步骤：

(1) 在用户空间堆栈中为信号处理程序的执行预先创建一个框架，框架中包括一个作为局部量的数据结构，并把系统空间堆栈中的“原始框架”保存到这个数据结构中。

(2) 在信号处理程序中插入对系统调用 sigreturn()的调用。

(3) 将系统空间堆栈中“原始框架”修改成为执行信号处理程序所需的框架。

(4) “返回”到用户空间，但是却执行信号处理程序。

(5) 信号处理程序执行完毕以后，通过系统调用 sigreturn()重返系统空间。

(6) 在系统调用 sigreturn()中从用户空间恢复“原始框架”。

(7) 再返回到用户空间，继续执行原先的用户程序。

知道了这个大致过程，有关的源代码就比较容易理解了。先来看文件 arch/i386/kernel/signal.c 中的函数 setup_frame()：

==================== arch/i386/kernel/signal.c 388 456 ====================

[ret_with_reschedule()>do_signal()>handle_signal()>setup_frame()]

```
static void setup_frame(int sig, struct k_sigaction *ka,
              sigset_t *set, struct pt_regs * regs)
{
    struct sigframe *frame;
    int err = 0;

    frame = get_sigframe(ka, regs, sizeof(*frame));

    if (!access_ok(VERIFY_WRITE, frame, sizeof(*frame)))
        goto give_sigsegv;

    err |= __put_user((current->exec_domain
               && current->exec_domain->signal_invmap
               && sig < 32
               ? current->exec_domain->signal_invmap[sig]
               : sig),
              &frame->sig);
    if (err)
        goto give_sigsegv;

    err |= setup_sigcontext(&frame->sc, &frame->fpstate, regs, set->sig[0]);
    if (err)
        goto give_sigsegv;

    if (_NSIG_WORDS > 1) {
        err |= __copy_to_user(frame->extramask, &set->sig[1],
                      sizeof(frame->extramask));
    }
    if (err)
        goto give_sigsegv;

    /* Set up to return from userspace.  If provided, use a stub
       already in userspace.  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	if (ka->sa.sa_flags & SA_RESTORER) {
		err |= __put_user(ka->sa.sa_restorer, &frame->pretcode);
	} else {
		err |= __put_user(frame->retcode, &frame->pretcode);
		/* This is popl %eax ; movl $,%eax ; int $0x80 */
		err |= __put_user(0xb858, (short *)(frame->retcode+0));
		err |= __put_user(__NR_sigreturn, (int *)(frame->retcode+2));
		err |= __put_user(0x80cd, (short *)(frame->retcode+6));
	}

	if (err)
		goto give_sigsegv;

	/* Set up registers for signal handler */
	regs->esp = (unsigned long) frame;
	regs->eip = (unsigned long) ka->sa.sa_handler;

	set_fs(USER_DS);
	regs->xds = __USER_DS;
	regs->xes = __USER_DS;
	regs->xss = __USER_DS;
	regs->xcs = __USER_CS;
	regs->eflags &= ~TF_MASK;

#if DEBUG_SIG
	printk("SIG deliver (%s:%d): sp=%p pc=%p ra=%p\n",
		current->comm, current->pid, frame, regs->eip, frame->pretcode);
#endif

	return;

give_sigsegv:
	if (sig == SIGSEGV)
		ka->sa.sa_handler = SIG_DFL;
	force_sig(SIGSEGV, current);
}
```

首先是用户空间中的框架，即 sigframe 数据结构，这也是在 arch/i386/kernel/signal.c 中定义的：

==================== arch/i386/kernel/signal.c 162 174 ====================

```
/*
 * Do a signal return; undo the signal stack.
 */

struct sigframe
{
	char *pretcode;
	int sig;
	struct sigcontext sc;
```

```
	struct _fpstate fpstate;
	unsigned long extramask[_NSIG_WORDS-1];
	char retcode[8];
};
```

这个数据结构实际上只是框架的一部分，因为具体的信号处理程序本身也会有其局部变量。所以，这个数据结构中的成分都是“附加”局部量，而对于信号处理程序来说是不可见的（在信号处理程序中不能引用这些局部量）。其中 sigcontext 数据结构的定义在 include/asm-i386/sigcontext.h 中：

==================== include/asm-i386/sigcontext.h 57 80 ====================

```
struct sigcontext {
	unsigned short gs, __gsh;
	unsigned short fs, __fsh;
	unsigned short es, __esh;
	unsigned short ds, __dsh;
	unsigned long edi;
	unsigned long esi;
	unsigned long ebp;
	unsigned long esp;
	unsigned long ebx;
	unsigned long edx;
	unsigned long ecx;
	unsigned long eax;
	unsigned long trapno;
	unsigned long err;
	unsigned long eip;
	unsigned short cs, __csh;
	unsigned long eflags;
	unsigned long esp_at_signal;
	unsigned short ss, __ssh;
	struct _fpstate * fpstate;
	unsigned long oldmask;
	unsigned long cr2;
};
```

显然，这个数据结构就是用来保存系统空间堆栈中的原始框架的。至于 sigframe 结构中其它成分的作用与用途，等一下就会清楚。框架的结构确定了，还要确定其在用户空间中的位置，这就是 get_sigframe()要做的事（arch/i386/kernel/signal.c）：

==================== arch/i386/kernel/signal.c 361 386 ====================

[ret_with_reschedule()>do_signal()>handle_signal()>setup_frame()>get_sigframe()]

```
/*
 * Determine which stack to use..
 */
static inline void *
get_sigframe(struct k_sigaction *ka, struct pt_regs * regs, size_t frame_size)
{
	unsigned long esp;

```

```
	/* Default to using normal stack */
	esp = regs->esp;

	/* This is the X/Open sanctioned signal stack switching.  */
	if (ka->sa.sa_flags & SA_ONSTACK) {
		if (! on_sig_stack(esp))
			esp = current->sas_ss_sp + current->sas_ss_size;
	}

	/* This is the legacy signal stack switching. */
	else if ((regs->xss & 0xffff) != __USER_DS &&
		 !(ka->sa.sa_flags & SA_RESTORER) &&
		 ka->sa.sa_restorer) {
		esp = (unsigned long) ka->sa.sa_restorer;
	}

	return (void *)((esp - frame_size) & -8ul);
}
```

这里的 regs->esp 是用户空间中当前的堆栈指针，也就是进入系统空间之前的堆栈指针，所以在典型情况下执行信号处理程序的框架就要从这一点往下伸展。但是，有两个例外。第一个例外是用户进程已经通过系统调用 sigaltstack()为信号处理程序的执行设置了替换堆栈，并且在设置“信号向量”时将 flags 中的标志位 SA_ONSTACK 设置成 1 这种情况下当前进程的 task_struct 结构中 sas_ss_sp 和 sas_ss_size 分别为所设置的堆栈位置和大小，而(sas_ss_sp + sas_ss_size)则为该堆栈空间的顶点，堆栈从这一点开始向下伸展。不过，先要检查一下是否已经在这个替换堆栈上。这里 inline 函数 on_sig_stack()的定义为（include/linux/sched.h）：

==================== include/linux/sched.h 631 636 ====================

[ret_with_reschedule()>do_signal()>handle_signal()>setup_frame()>get_sigframe()>on_sig_stack()]

```
/* True if we are on the alternate signal stack.  */

static inline int on_sig_stack(unsigned long sp)
{
	return (sp - current->sas_ss_sp < current->sas_ss_size);
}
```

第二个例外与在执行完信号处理程序后重返系统空间的过程有关。如前所述，最妥当的办法是让内核自动插入一些代码，通过系统调用 sigreturn()解决这个问题。也就是说，把这个问题交给操作系统，用户进程就不用操这个心了。可是在发展的过程中有过一段时期，曾经在系统调用 sigaction()的界面上提供了一个手段，让用户给定一段程序用于这个目的，这就是 sigaction 数据结构中的指针 sa_restorer。现在，系统调用 sigaction()的“ man”页面中已经明确讲 sa_restorer“已经过时”并且“不应使用”，但是出于兼容的需要还得考虑其存在。所以，如果使用了 sa_restorer，就要把框架的顶点设置在这个位置上。

至此，框架的顶点已经确定了。由于堆栈是向下伸展的，所以这个框架（其实是框架的一部分）的起始地址在其下方相差一个 frame_size 的地方，而 frame_size 正是 sigframe 数据结构的大小。注意这

里的无符号长整数-8 实际上是 0xffff fff8，用以对齐框架起始地址的边界。这样，对于信号处理程序，这个 sigframe 数据结构就相当于一个额外的调用参数。

用户空间框架的位置 frame 已经确定，让我们回到 setup_frame()中。接着检验一下用户空间中的这一部分是否可写，然后就是往用户空间的这个框架中复制信息了。这里的__put_user()将其第一个参数复制到用户空间中由其第二个参数所指向的地方。首先是 frame->sig，因为这个量有点特殊。在有的“执行域”，即 Unix 变种里（见第 4 章中有关内容），为信号定义的数值可能会有所不同。在这种情况下相应 exec_domain 数据结构中的指针 signal_invmap 会指向一个信号变换表，所以要把这个因素考虑进去。下面就是复制系统空间堆栈上的 pt_regs 结构以及一些有关的内容了，包括有关浮点处理器的内容和信号屏蔽位图（arch/i386/kernel/signal.c）：

==================== arch/i386/kernel/signal.c 314 359 ====================

[ret_with_reschedule()>do_signal()>handle_signal()>setup_frame()>setup_sigcontext()]

```
/*
 * Set up a signal frame.
 */

static int
setup_sigcontext(struct sigcontext *sc, struct _fpstate *fpstate,
           struct pt_regs *regs, unsigned long mask)
{
     int tmp, err = 0;

     tmp = 0;
     __asm__("movl %%gs,%0" : "=r"(tmp): "0"(tmp));
     err |= __put_user(tmp, (unsigned int *)&sc->gs);
     __asm__("movl %%fs,%0" : "=r"(tmp): "0"(tmp));
     err |= __put_user(tmp, (unsigned int *)&sc->fs);

     err |= __put_user(regs->xes, (unsigned int *)&sc->es);
     err |= __put_user(regs->xds, (unsigned int *)&sc->ds);
     err |= __put_user(regs->edi, &sc->edi);
     err |= __put_user(regs->esi, &sc->esi);
     err |= __put_user(regs->ebp, &sc->ebp);
     err |= __put_user(regs->esp, &sc->esp);
     err |= __put_user(regs->ebx, &sc->ebx);
     err |= __put_user(regs->edx, &sc->edx);
     err |= __put_user(regs->ecx, &sc->ecx);
     err |= __put_user(regs->eax, &sc->eax);
     err |= __put_user(current->thread.trap_no, &sc->trapno);
     err |= __put_user(current->thread.error_code, &sc->err);
     err |= __put_user(regs->eip, &sc->eip);
     err |= __put_user(regs->xcs, (unsigned int *)&sc->cs);
     err |= __put_user(regs->eflags, &sc->eflags);
     err |= __put_user(regs->esp, &sc->esp_at_signal);
     err |= __put_user(regs->xss, (unsigned int *)&sc->ss);

     tmp = save_i387(fpstate);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
349        if (tmp < 0)
350          err = 1;
351        else
352          err |= __put_user(tmp ? fpstate : NULL, &sc->fpstate);
353
354        /* non-iBCS2 extensions.. */
355        err |= __put_user(mask, &sc->oldmask);
356        err |= __put_user(current->thread.cr2, &sc->cr2);
357
358        return err;
359   }
```

我们把这个函数的代码留给读者。完成以后，如果信号屏蔽位图的大小超过一个长整数的大小，则还要把超出的部分也复制过去。

下面是关键的部分，也就是对重返系统空间进行安排了。在 sigframe 数据结构中有个 8 字节的数组 retcode[]，还有个指针 pretcode。指针 pretcode 指向一段使进程在执行完信号处理程序后重返系统空间的代码。如果用户提供了这么一个函数，就把该函数指针复制到 pretcode 中；否则，在典型的情况下，就使这个指针指向 retcode[]（见 424 行），并且在 retcode[]中预先写入这样三条指令（见 426～428 行）：

```
popl %eax;
movl $__NR_sigreturn, %eax;
int $0x80;
```

这三条指令正好占 8 个字节。指令中的__NR_sigreturn 为系统调用 sigreturn()的调用号。经过这样处理以后，用户空间中堆栈的构成如图 6.7 所示。

这里要指出，当前进程返回到用户空间时（下面就会看到），是“返回”而不是“调用”进入信号处理程序的，所以在 pretcode 的下方不会再压入一个“返回地址”。这样，pretcode 就正好处在本来应该是信号处理程序运行框架中返回地址的位置上。在信号处理程序的末尾执行 ret 指令时，就会把它当成返回地址而转入预先“埋伏”在 retcode[]中的三条指令，或者由用户另行提供的 sa_restorer 函数。而位于 pretcode 上方的 sig，则成为对信号处理程序的第一个调用参数。可见，sigframe 数据结构的内容，包括各个字段的次序，是根据整个执行过程精心设计好的，不能随便更改。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

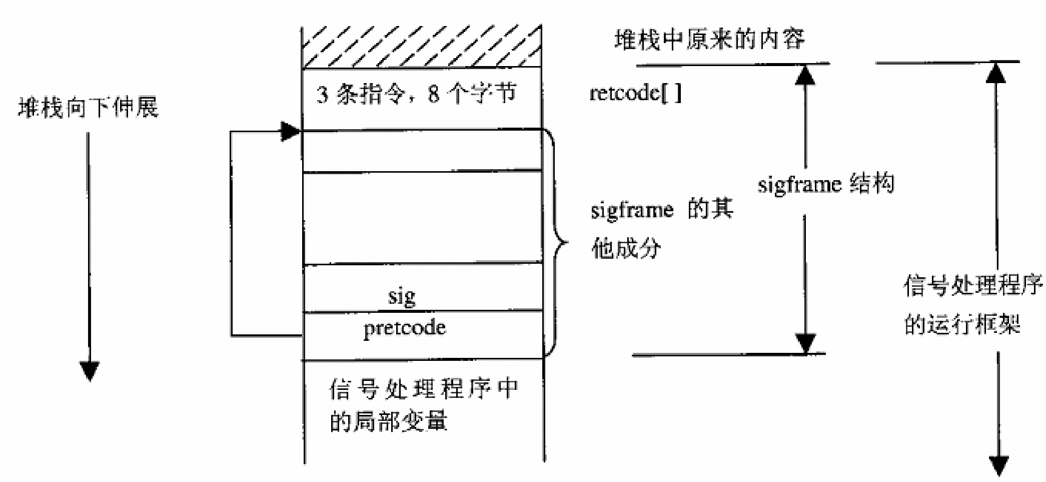


图 6.7　信号处理结束返回系统空间前用户空间堆栈结构图

读者也许要问：这里一共就是三条指令，进程在执行“ int $0x80”指令进入 sigreturn()系统调用之后最终还要回到用户空间来，那时候按理应该回到它的下一条指令，可是这里没有指令了啊。答案是，在系统调用 sigreturn()中，要从用户空间的这个框架中恢复转入用户空间之前的“原始框架”，所以到那时候就会返回到原先应该去的地方，也就是当初发生中断、异常或系统调用的地方。

安排好了用户空间中的框架，就要安排系统空间的框架了。这里最关键的是返回到用户空间时的堆栈指针 regs->esp 和取指令指针 regs->eip。还有就是一些段寄存器，不过其实这些寄存器的值本来也就是__USER_DS 和__USER_CS，只有在特殊的情况下才有例外。至于一些通用寄存器，如%eax，%ebx 等，则对于信号处理程序并无意义，所以不需要设置。最后，处理器在用户空间时有可能正处于硬件跟踪模式，而信号处理程序相应于中断处理，所以在进入这段程序时要把硬件跟踪关闭，也就是在标志寄存器的映象 regs->eflags 中将 TF 位清 0。经过这样的安排以后，就为表面上按正常途径“返回 ”用户空间，但是实际上却转入信号处理程序作好了准备。最终，处理器经由 setup_frame()，handle_signal()和 do_signal()逐层返回到 entry.S 中的 signal_return，从而进入 restore_all。从那以后的过程读者应该已经熟悉了（见第 3 章）。由于 regs->esp 和 regs->eip 的设置，处理器进入用户空间时从 ka->sa.sa_handler 所指向的地方开始执行，而堆栈指针则指向前面设置好了的框架，实际上是指向 frame->pretcode，即信号处理程序的返回地址，正好跟在用户空间中通过 call 语句进入信号处理程序时一样。

信号处理程序执行完毕以后，处理器又通过系统调用 sigreturn()进入系统空间。内核中实现这个系统调用的主体是 sys_sigreturn()，其代码也在 arch/i386/kernel/signal.c 中：

==================== arch/i386/kernel/signal.c 249 277 ====================

```
asmlinkage int sys_sigreturn(unsigned long __unused)
{
	struct pt_regs *regs = (struct pt_regs *) &__unused;
	struct sigframe *frame = (struct sigframe *)(regs->esp - 8);
	sigset_t set;
	int eax;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (verify_area(VERIFY_READ, frame, sizeof(*frame)))
        goto badframe;
    if (__get_user(set.sig[0], &frame->sc.oldmask)
       || (_NSIG_WORDS > 1
        && __copy_from_user(&set.sig[1], &frame->extramask,
                     sizeof(frame->extramask))))
        goto badframe;

    sigdelsetmask(&set, ~_BLOCKABLE);
    spin_lock_irq(&current->sigmask_lock);
    current->blocked = set;
    recalc_sigpending(current);
    spin_unlock_irq(&current->sigmask_lock);

    if (restore_sigcontext(regs, &frame->sc, &eax))
        goto badframe;
    return eax;

badframe:
    force_sig(SIGSEGV, current);
    return 0;
}
```

显然，这段程序的作用，就是从用户空间执行信号处理程序的框架中恢复当初系统空间中的原始框架。我们把这段程序留给读者自己阅读，不过有两点要提示一下。

首先，系统调用的框架就是系统空间堆栈上的 pt_regs 数据结构。在 sys_sigreturn()中取第一个调用参数__unused 的地址就得到了这个结构的起始地址。读者不妨回顾一下第 3 章中的有关内容。在执行完宏操作 SAVE_ALL 以后，系统空间堆栈中的最后一项，也就是 pt_regs 结构中的第一项，是%ebx 的内容。它的下面就是调用 sys_sigreturn()的第一个（也是惟一的）参数__unused。不过这里并不需要用到这个参数的内容，而只是要知道它在堆栈中的地址，因为这就是 pt_regs 数据结构的起始地址。

还有，就是用户空间中的框架，也就是 sigframe 数据结构的起始地址 frame。该结构中底部的第一项 pretcode 就是信号处理程序的返回地址，所以当处理器从信号处理程序返回时，堆栈指针就调整到了这一项的上方，也就是起始地址加 4 个字节的地方。然后，前述三条指令中的第一条“ popl %eax ”又使堆栈指针往上调了 4 个字节。这样，当处理器在用户空间执行 int 指令进入系统空间时，其用户空间的堆栈指针指向该 sigframe 结构的起始地址再加 8 个字节的地方，所以（regs->esp-8）就是这个结构的起始地址。

函数中其余的代码，以及处理器使用恢复后的原始框架返回用户空间的过程，读者应该不会有什么困难了。

读者也许要问，既然通过 sigreturn()重返系统空间以后实际上不干什么事，只是恢复了原始框架以后就从“原先的系统调用（或中断）”返回了，那么是否可以简化一点呢？例如，可以在用户空间堆栈中，从当前的系统调用框架向下调整，先将系统空间堆栈中的返回地址搬到用户空间堆栈中，而把系统空间堆栈中的返回地址，改成指向用户空间的信号处理程序。这样，从当前系统调用返回时就会返回到用户空间中的信号处理程序。而在执行完信号处理程序后碰到 ret 指令时，则又返回到原先进行系

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

统调用或发生中断的地方。这样，整个过程简化了，代码也简单了，而系统调用 sigreturn()也不需要了，岂不很好？事实上，早期的 unix（如第 6 版）正是这样做的。但是在这样的解决方案中必须有个保证，就是用户空间的信号处理程序对于处理器的“工作现场”（即内核中通过 SAVE_ALL 保存的所有寄存器的内容）完全“透明”，即不改变这些寄存器的内容。例如，如果能保证在进入信号处理程序时一定会调用一段类似于 SAVE_ALL 的程序，而在离开信号处理程序之前则调用一段类似于 RESTORE_ALL 的程序，那就没有问题了。然而，信号处理程序是由用户开发，且在用户空间中运行的，没有一个通用有效并且可靠的方法可以保证由用户开发的程序对寄存器内容的透明性。明白了这一点，就可以理解为什么要煞费苦心地来设计一个 sigreturn()系统调用了。

通过对从设置“信号向量”、发送信号、到执行信号处理程序的全过程的了解，并且将此过程与中断机制中设置“中断向量”、中断请求、到执行中断处理程序的过程加以类比，读者应该对信号机制有了比较深入的理解。当然，进程之间通过信号机制的互动要有用户程序的参与，而那已是属于应用程序设计的范畴了。有兴趣（或者有需要）的读者可以参考有关专著。

6.5　系统调用 ptrace()和进程跟踪

为方便应用软件的开发和调试，从 Unix 的早期版本开始就提供了一种对运行中的进程进行跟踪和控制的手段，那就是系统调用 ptrace()。通过 ptrace()，一个进程可以动态地读 / 写另一个进程的内存和寄存器，包括其指令空间、数据空间、堆栈以及所有的寄存器。与信号机制（以及其它手段）相结合，就可以实现让一个进程在另一个进程的控制和跟踪下运行的目的。GNU 的调试工具 gdb 就是一个典型的实例。通过 gdb，软件开发人员可以使一个应用程序在 gdb 的“监视”和操纵下受控地运行。对于受 gdb 控制的进程，可以通过在其程序中设置断点，检查其堆栈以确定函数调用路线，检查并改变局部变量或全局变量的内容等等方法，来进行调试。显然，所有这些手段从概念上说都确实属于进程间“通信”的范畴，但是必须指出，这只是为软件调试而设计和设立的，不应该用于一般的进程间通信。一般而言，通信是要由双方都介入且互相协调才能完成的。就拿“管道”来说，虽然管道是单向的，但一定得由一方写，另一方读才能达到目的。再拿信号来说，虽然信号是异步的，也就是接收信号的一方并不知道信号会在什么时候到来，因而在应用程序中并不主动有意地去检查有否信号到达。但是从总体而言，接收方知道信号可能会到来，并且为此在应用程序中作出了安排。而当信号真的到来时，接收方也“知道”其到来，并根据事先的安排作出反应。然而，由 ptrace()所实现的“通信”却完全是单方面的，被跟踪的进程甚至并不知道（从应用程序的角度而言）自己是在受到控制和监视的条件下运行。从这个角度讲，ptrace()其实又不属于“进程间通信”。

那么，怎样通过 ptrace()来实现上述这些目的呢？先来看看这个系统调用的格式：

```
int ptrace(int request, int pid, int addr, int data);
```

参数 pid 为进程号，指明了操作的对象，而 request，则是具体的操作，文件 include/linux/ptrace.h 中定义了所有可能的操作码：

==================== include/linux/ptrace.h 8 22 ====================

```
8    #define PTRACE_TRACEME         0
```

```
#define PTRACE_PEEKTEXT        1
#define PTRACE_PEEKDATA        2
#define PTRACE_PEEKUSR         3
#define PTRACE_POKETEXT        4
#define PTRACE_POKEDATA        5
#define PTRACE_POKEUSR         6
#define PTRACE_CONT            7
#define PTRACE_KILL            8
#define PTRACE_SINGLESTEP      9

#define PTRACE_ATTACH       0x10
#define PTRACE_DETACH       0x11

#define PTRACE_SYSCALL        24
```

跟踪者（如 gdb）先要通过 PTRACE_ATTACH 与被跟踪进程建立起关系，或者说“Attach”到被跟踪进程。然后，就可以通过各种 PEEK 和 POKE 操作来读 / 写被跟踪进程的指令空间、数据空间或者各个寄存器，每次都是一个长字，由 addr 指明其地址；或者，也可以通过 PTRACE_SINGLESTEP、PTRACE_KILL，PTRACE_SYSCALL 和 PTRACE_CONT 等操作来控制被跟踪进程的运行。最后，通过 PTRACE_DETACH 跟被跟踪进程脱离关系。所有这些操作都是单方面的，被跟踪进程既不能拒绝，也无需“合作”。惟一例外是 PTRACE_TRACEME，用来主动接受跟踪。

像其它系统调用一样，ptrace()在内核中的实现是 sys_ptrace()。其代码在 arch/i386/kernel/ptrace.c 中：

==================== arch/i386/kernel/ptrace.c 137 153 ====================

[sys_ptrace()]

```
asmlinkage int sys_ptrace(long request, long pid, long addr, long data)
{
    struct task_struct *child;
    struct user * dummy = NULL;
    int i, ret;

    lock_kernel();
    ret = -EPERM;
    if (request == PTRACE_TRACEME) {
        /* are we already being traced? */
        if (current->ptrace & PT_PTRACED)
            goto out;
        /* set the ptrace bit in the process flags. */
        current->ptrace |= PT_PTRACED;
        ret = 0;
        goto out;
    }
```

首先是对 PTRACE_TRACEME 的处理，那就是设置当前进程 task_struct 中的标志位 PF_PTRACED。这个标志位的作用读者以后会看到。如果不是主动请求跟踪，那就一定有个目标进程了。继续往下看代码：

==================== arch/i386/kernel/ptrace.c 154 166 ====================

[sys_ptrace()]

```
154        ret = -ESRCH;
155        read_lock(&tasklist_lock);
156        child = find_task_by_pid(pid);
157        if (child)
158            get_task_struct(child);
159        read_unlock(&tasklist_lock);
160        if (!child)
161            goto out;
162
163        ret = -EPERM;
164        if (pid == 1)        /* you may not mess with init */
165            goto out_tsk;
166
```

函数 find_task_by_pid()，顾名思义就是根据进程号找到目标进程的 task_struct 结构。可是跟踪者怎样才能知道目标进程的进程号呢？以 gdb 为例有两种情况。一种情况是被跟踪进程本来就是由 gdb 通过 fork()和 exec()启动的。这种情况下的命令行为：

gdb prog

执行 prog 的进程本来就是 gdb 的子进程，所以 gdb 当然知道它的进程号。

另一种情况是 prog 进程在启动 gdb 之前已经在运行。在这种情况下操作人员要先弄清楚它的进程号（如通过“ps”），再把这进程号作为参数用在启动 gdb 的命令行中。此时的命令行类似于：

gdb prog 1234

因此，在这两种情况下 gdb 都会知道目标进程的进程号。

不过请注意，1 号进程，即初始化进程 init()是不允许跟踪的。

找到目标进程以后，要通过 get_task_struct()递增对子进程的 task_struct 所在页面的使用计数，到完成了操作以后再通过后面（468 行）的 free_task_struct()还原。这是因为有些操作在过程中可能会发生进程调度（读者可以自己看一下 access_process_vm()的代码），需要防止因为子进程先得到机会运行并且 exit()，从而将其 task_struct 结构所在页面释放掉的可能。

现在可以执行具体的操作了，先来看 PTRACE_ATTACH：

==================== arch/i386/kernel/ptrace.c 167 195 ====================

[sys_ptrace()]

```
167        if (request == PTRACE_ATTACH) {
168            if (child == current)
169                goto out_tsk;
170            if ((!child->dumpable ||
171                (current->uid != child->euid) ||
172                (current->uid != child->suid) ||
173                (current->uid != child->uid) ||
174                (current->gid != child->egid) ||
175                (current->gid != child->sgid) ||
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
176                 (!cap_issubset(child->cap_permitted, current->cap_permitted)) ||
177                 (current->gid != child->gid)) && !capable(CAP_SYS_PTRACE))
178                  goto out_tsk;
179             /* the same process cannot be attached many times */
180             if (child->ptrace & PT_PTRACED)
181                  goto out_tsk;
182             child->ptrace |= PT_PTRACED;
183 
184             write_lock_irq(&tasklist_lock);
185             if (child->p_pptr != current) {
186                  REMOVE_LINKS(child);
187                  child->p_pptr = current;
188                  SET_LINKS(child);
189             }
190             write_unlock_irq(&tasklist_lock);
191 
192             send_sig(SIGSTOP, child, 1);
193             ret = 0;
194             goto out_tsk;
195        }
```

跟踪不是无条件的。谁可以跟踪谁，需要满足一些条件。首先，自己不允许（也不必要）跟踪自己。除此之外，170 行开始的 if 语句给出了这些条件。一般来说，两个进程要属于同一用户或同一组。读者可以参看“文件系统”一章中的有关内容，搞清这些条件的含意。注意这里的 capable()定义为 suser()，也就是说如果两个进程不属于同一组，就要将当前进程提升为特权用户进程才行，而这当然也是有条件的。此外，被跟踪的进程必须是尚未受其它进程跟踪的。所谓 attach，或者说建立起跟踪关系，就是做三件事：一是将被跟踪进程的 PF_TRACED 标志设成 1（182 行）。还有，就是如果被跟踪进程不是跟踪者的子进程就将其“收养”为跟踪者的子进程（185～189 行）。最后，还要向被跟踪进程发送一个 SIGSTOP 信号（192 行），这样被跟踪进程被调度运行时就会对信号作出反应而进入暂停状态。

如果不是 PTRACE_ATTACH 话，那就必然是对已经处于被跟踪地位的进程的后续操作了。

==================== arch/i386/kernel/ptrace.c 196 204 ====================

[sys_ptrace()]

```
196        ret = -ESRCH;
197        if (!(child->ptrace & PT_PTRACED))
198             goto out_tsk;
199        if (child->state != TASK_STOPPED) {
200             if (request != PTRACE_KILL)
201                  goto out_tsk;
202        }
203        if (child->p_pptr != current)
204             goto out_tsk;
```

就是说，先要加以核实。目标进程的 PF_TRACED 标志位必须是 1，目标进程必须是当前进程的子进程，并且处于 TASK_STOPPED 状态。这也说明，如果目标进程是通过 PTRAC_TRACEME 操作主动接受跟踪的话，只有其父进程才能对其实行跟踪，并且先要向其发送一个 SIGSTOP 信号。所以，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

跟踪只能对子进程进行，哪怕是临时“收养”的子进程。

通过了对条件的检验以后，就进入一个 switch 语句，针对不同的操作码来执行了：

==================== arch/i386/kernel/ptrace.c 205 219 ====================

[sys_ptrace()]

```
	switch (request) {
	/* when I and D space are separate, these will need to be fixed. */
	case PTRACE_PEEKTEXT: /* read word at location addr. */
	case PTRACE_PEEKDATA: {
		unsigned long tmp;
		int copied;

		copied = access_process_vm(child, addr, &tmp, sizeof(tmp), 0);
		ret = -EIO;
		if (copied != sizeof(tmp))
			break;
		ret = put_user(tmp,(unsigned long *) data);
		break;
	}

```

PTRACE_PEEKTEXT 操作从子进程的指令空间，或称代码段中地址为 addr 处读取一个长字，而 PTRACE_PEEKDATA 则从子进程的数据空间读一个长字。读者可以回顾一下第 2 章中有关的内容，在 Linux 内核中代码空间和数据空间实际上是一致的，所以二者可以合并在一起处理。函数 access_process_vm()的代码在 kernel/ptrace.c 中，我们把它作为对第 2 章的复习，留给读者自己阅读。access_process_vm()是个对给定进程的存储空间进行读或写的通用函数。它先通过 find_extend_vma()找到该进程包含着给定地址的虚存区间，然后根据需要读/写的长度在 access_mm()中通过 access_one_page()访问所涉及的各个页面。而 access_one_page()则是对给定进程的某一页面进行读写的通用函数，它从进程的某一虚存区间，也就是 vm_area_struct 结构开始，先找到给定页面所在的页面目录项，然后往下找到相应的页面项。找到页面项以后，就可以将其所映射的物理页面临时映射到当前进程的虚存空间中，并对其进行读/写。完成以后，就通过 put_user()将读取的长字写回当前进程的用户空间。也许有读者会问，当父进程正在读子进程的物理空间时，会不会子进程也正好在同一地址上写，从而使读出的数据不正确呢？不会的。首先，前面的检验已经确保了子进程正处于“暂停”状态 TASK_STOPPED（这已经是考虑到了多处理器的情况，至于在单处理器系统中，则既然当前进程正在运行，其子进程显然不在运行）。另外，对 PTRACE_PEEKTEXT 和 PTRACE_PEEKDATA 而言，所读取的只是一个长字，在 32 位的 CPU 中只要一条指令就完成了，是个“原子”操作。

回顾一下第 2 章和第 3 章，读者就可以明白，进程的用户空间堆栈也在其数据空间中，所以也可以通过 PTRAC_PEEKDATA 操作来读子进程用户空间的堆栈。当然，先要通过其它操作得到其用户空间的堆栈指针。

最后，还要指出，PTRACE_PEEKDATA 和 PTRACE_PEEKTEXT 只能用来读子进程的用户空间，而不能用来读系统（内核）空间，这是由函数 find_extend_vma()所保证的。但是，子进程的（与跟踪有关的）有些信息却在系统空间中。例如，当子进程处于睡眠或暂停状态时，其进入系统空间前夕的寄存器内容都保存在它的系统空间堆栈中（pt_regs 结构），还有些信息则在它的 task_struct 结构内部的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

一个 thread_struct 结构中。怎样读取这些信息呢？沿着 ptrace.c 的代码继续往下看：

==================== arch/i386/kernel/ptrace.c 220 241 ====================

[sys_ptrace()]

```
	/* read the word at location addr in the USER area. */
	case PTRACE_PEEKUSR: {
		unsigned long tmp;

		ret = -EIO;
		if ((addr & 3) || addr < 0 ||
		    addr > sizeof(struct user) - 3)
			break;

		tmp = 0;  /* Default return condition */
		if(addr < 17*sizeof(long))
			tmp = getreg(child, addr);
		if(addr >= (long) &dummy->u_debugreg[0] &&
		   addr <= (long) &dummy->u_debugreg[7]){
			addr -= (long) &dummy->u_debugreg[0];
			addr = addr >> 2;
			tmp = child->thread.debugreg[addr];
		}
		ret = put_user(tmp,(unsigned long *) data);
		break;
	}

```

这个操作有两种作用，第一是用于读取子进程在用户空间运行时（进入系统空间前夕）的某个寄存器的内容（注意此时子进程必定在系统空间中，因为调度和切换只发生于系统空间）。我们先来看这一部分。要读取一个寄存器的内容时，参数 addr 必须是寄存器号乘以 4。对 i386 处理器而言共有 17 个这样的寄存器，定义于 ptrace.h 中。不过，所谓“寄存器”其实并不完全是字面意义上的，例如 EAX 和 ORIG_EAX 就算作两项，因为在系统空间堆栈的 pt_regs 结构中它们是有区别的(系统调用使用 EAX 来返回出错代码)。当 addr 指明这 17 个“寄存器”之一时，就通过 getreg()来读取其内容（代码在同一文件 arch/i386/kernel/ptrace.c 中）：

==================== arch/i386/kernel/ptrace.c 110 135 ====================

[sys_ptrace()>getreg()]

```
static unsigned long getreg(struct task_struct *child,
	unsigned long regno)
{
	unsigned long retval = ~0UL;

	switch (regno >> 2) {
		case FS:
			retval = child->thread.fs;
			break;
		case GS:
			retval = child->thread.gs;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
121                 break;
122             case DS:
123             case ES:
124             case SS:
125             case CS:
126                 retval = 0xffff;
127                 /* fall through */
128             default:
129                 if (regno > GS*4)
130                     regno -= 2*4;
131                 regno = regno - sizeof(struct pt_regs);
132                 retval &= get_stack_long(child, regno);
133         }
134         return retval;
135     }
```

也就是说，除 FS 和 GS 的映象在 thread_struct 结构中外，其余的都在系统空间堆栈的 pt_regs 结构中。注意，第 127 行处并无 break 语句。函数 get_stack_long()的代码也在同一文件中：

==================== arch/i386/kernel/ptrace.c 41 54 ====================
[sys_ptrace()>getreg()>get_stack_long()]

```
/*
 * this routine will get a word off of the processes privileged stack.
 * the offset is how far from the base addr as stored in the TSS.
 * this routine assumes that all the privileged stacks are in our
 * data space.
 */
static inline int get_stack_long(struct task_struct *task, int offset)
{
    unsigned char *stack;

    stack = (unsigned char *)task->thread.esp0;
    stack += offset;
    return (*((int *)stack));
}
```

读者也许还记得，一个进程的 thread_struct 结构中的的 esp0 保存着其系统空间堆栈指针。当进程穿过中断门、陷阱门或调用门进入系统空间时，处理器会从这里恢复其系统空间堆栈。

再来看 PTRACE_PEEKUSR 的第二种作用，这就要先介绍一些背景知识了。Intel 在 i386 系统结构中首创性地引入了“调试寄存器”（debug registers），为软件的开发与维护提供了功能很强而且效率很高的调试手段。用户进程可以通过设置一些调试寄存器来使处理器在一定的条件下落入“陷阱”，从而进入一个“断点”，即一段调试程序。这些条件包括：①当处理器执行到某一指令时；②当处理器读某一内存地址时；③从处理器写某一内存地址时。而“陷阱”则是指专门用于虚地址模式程序调试的 1 号陷阱 debug（另有一个用于实地址模式的 3 号陷阱 int 3，在 Linux 中仅用于 vm86 模式）。内核中对这个陷阱的处理程序为 do_debug()，其代码在 arch/i386/kernel/traps.c 中。有关调试寄存器的详情则请参阅 Intel 的手册或其它技术资料：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== arch/i386/kernel/traps.c 488 575 ====================

```
/*
 * Our handling of the processor debug registers is non-trivial.
 * We do not clear them on entry and exit from the kernel. Therefore
 * it is possible to get a watchpoint trap here from inside the kernel.
 * However, the code in ./ptrace.c has ensured that the user can
 * only set watchpoints on userspace addresses. Therefore the in-kernel
 * watchpoint trap can only occur in code which is reading/writing
 * from user space. Such code must not hold kernel locks (since it
 * can equally take a page fault), therefore it is safe to call
 * force_sig_info even though that claims and releases locks.
 *
 * Code in ./signal.c ensures that the debug control register
 * is restored before we deliver any signal, and therefore that
 * user code runs with the correct debug control register even though
 * we clear it here.
 *
 * Being careful here means that we don't have to be as careful in a
 * lot of more complicated places (task switching can be a bit lazy
 * about restoring all the debug state, and ptrace doesn't have to
 * find every occurrence of the TF bit that could be saved away even
 * by user code)
 */
asmlinkage void do_debug(struct pt_regs * regs, long error_code)
{
     unsigned int condition;
     struct task_struct *tsk = current;
     siginfo_t info;

     __asm__ __volatile__("movl %%db6,%0" : "=r" (condition));

     /* Mask out spurious debug traps due to lazy DR7 setting */
     if (condition & (DR_TRAP0|DR_TRAP1|DR_TRAP2|DR_TRAP3)) {
          if (!tsk->thread.debugreg[7])
               goto clear_dr7;
     }

     if (regs->eflags & VM_MASK)
          goto debug_vm86;

     /* Save debug status register where ptrace can see it */
     tsk->thread.debugreg[6] = condition;

     /* Mask out spurious TF errors due to lazy TF clearing */
     if (condition & DR_STEP) {
          /*
           * The TF error should be masked out only if the current
           * process is not traced and if the TRAP flag has been set
           * previously by a tracing process (condition detected by
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
536               * the PT_DTRACE flag); remember that the i386 TRAP flag
537               * can be modified by the process itself in user mode,
538               * allowing programs to debug themselves without the ptrace()
539               * interface.
540               */
541              if ((tsk->ptrace & (PT_DTRACE|PT_PTRACED)) == PT_DTRACE)
542                   goto clear_TF;
543         }
544
545         /* Ok, finally something we can handle */
546         tsk->thread.trap_no = 1;
547         tsk->thread.error_code = error_code;
548         info.si_signo = SIGTRAP;
549         info.si_errno = 0;
550         info.si_code = TRAP_BRKPT;
551
552         /* If this is a kernel mode trap, save the user PC on entry to
553          * the kernel, that's what the debugger can make sense of.
554          */
555         info.si_addr = ((regs->xcs & 3) == 0) ? (void *)tsk->thread.eip :
556                                                  (void *)regs->eip;
557         force_sig_info(SIGTRAP, &info, tsk);
558
559         /* Disable additional traps. They'll be re-enabled when
560          * the signal is delivered.
561          */
562    clear_dr7:
563         __asm__("movl %0,%%db7"
564              : /* no output */
565              : "r" (0));
566         return;
567
568    debug_vm86:
569         handle_vm86_trap((struct kernel_vm86_regs *) regs, error_code, 1);
570         return;
571
572    clear_TF:
573         regs->eflags &= ~TF_MASK;
574         return;
575    }
```

我们不详细讲解这些程序了，但是读者可以看到它对当前进程，也就是引起此次陷阱的进程发出一个 SIGTRAP 信号（见 557 行），并且通过 siginfo_t 数据结构载送断点所在的地址（见 555 行）。当然，引起这次陷阱的进程要事先为处理这个信号作好准备（否则进程就会“流产”）。这也是为什么在编译供调试的程序时要使用“ -g”选择项的原因之一。

回到 PTRACE_PEEKUSR 的代码中。这里的局部量 dummy 是个 user 结构指针，其值在开头时初始化成 NULL。第 232 和 233 行是对 addr 的范围进行检查。也就是，假定一个 user 结构是从地址 0 开

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

始的，看 addr 的值是否对应于该结构中 u_debugreg[]数组的偏移量。数据结构 struct_user 是在进程“流产”（abort）时转储（dump）内存映象时使用的，定义于 include/asm-i386/user.c 中：

==================== include/asm-i386/user.h 88 116 ====================

```
/* When the kernel dumps core, it starts by dumping the user struct -
   this will be used by gdb to figure out where the data and stack segments
   are within the file, and what virtual addresses to use. */
struct user{
/* We start with the registers, to mimic the way that "memory" is returned
   from the ptrace(3,...) function.  */
  struct user_regs_struct regs;       /* Where the registers are actually stored */
/* ptrace does not yet supply these.  Someday.... */
  int u_fpvalid;        /* True if math co-processor being used. */
                            /* for this mess. Not yet used. */
  struct user_i387_struct i387;  /* Math Co-processor registers. */
/* The rest of this junk is to help gdb figure out what goes where */
  unsigned long int u_tsize;/* Text segment size (pages). */
  unsigned long int u_dsize;/* Data segment size (pages). */
  unsigned long int u_ssize;/* Stack segment size (pages). */
  unsigned long start_code;     /* Starting virtual address of text. */
  unsigned long start_stack;/* Starting virtual address of stack area.
                       This is actually the bottom of the stack,
                       the top of the stack is always found in the
                       esp register.  */
  long int signal;           /* Signal that caused the core dump. */
  int reserved;              /* No longer used */
  struct user_pt_regs * u_ar0;   /* Used by gdb to help find the values for */
                    /* the registers. */
  struct user_i387_struct* u_fpstate;/* Math Co-processor pointer. */
  unsigned long magic;       /* To uniquely identify a core file */
  char u_comm[32];      /* User command that was responsible */
  int u_debugreg[8];
};
```

不过，这个数据结构在这里只是用来检查参数 addr 的范围，而具体调试寄存器的映象则在子进程的 thread_struct 结构中（见 236 行）。

PTRACE_POKETEXT 和 PTRACE_POKEDATA 是前面两个操作的逆向操作，代码很简单（arch/i386/kernel/ptrace.c）。

==================== arch/i386/kernel/ptrace.c 242 250 ====================

[sys_ptrace()]

```
    /* when I and D space are separate, this will have to be fixed. */
    case PTRACE_POKETEXT: /* write the word at location addr. */
    case PTRACE_POKEDATA:
        ret = 0;
        if (access_process_vm(child, addr, &data, sizeof(data), 1) == sizeof(data))
            break;
        ret = -EIO;
```

```
        break;

```

PTRACE_POKEUSR 则稍微要复杂一点：

==================== arch/i386/kernel/ptrace.c 251 288 ====================

[sys_ptrace()]

```
    case PTRACE_POKEUSR: /* write the word at location addr in the USER area */
        ret = -EIO;
        if ((addr & 3) || addr < 0 ||
            addr > sizeof(struct user) - 3)
            break;

        if (addr < 17*sizeof(long)) {
            ret = putreg(child, addr, data);
            break;
        }
        /* We need to be very careful here.  We implicitly
           want to modify a portion of the task_struct, and we
           have to be selective about what portions we allow someone
           to modify. */

          ret = -EIO;
          if(addr >= (long) &dummy->u_debugreg[0] &&
             addr <= (long) &dummy->u_debugreg[7]){

              if(addr == (long) &dummy->u_debugreg[4]) break;
              if(addr == (long) &dummy->u_debugreg[5]) break;
              if(addr < (long) &dummy->u_debugreg[4] &&
                 ((unsigned long) data) >= TASK_SIZE-3) break;

              if(addr == (long) &dummy->u_debugreg[7]) {
                  data &= ~DR_CONTROL_RESERVED;
                  for(i=0; i<4; i++)
                      if ((0x5f54 >> ((data >> (16 + 4*i)) & 0xf)) & 1)
                          goto out_tsk;
              }

              addr -= (long) &dummy->u_debugreg;
              addr = addr >> 2;
              child->thread.debugreg[addr] = data;
              ret = 0;
          }
          break;

```

这里的特别之处仅在于对参数 addr 和 data 的检查。首先，调试寄存器 0 至 3 这四个寄存器是允许设置的，但是要检查所设置的值 data（实际上是个内存地址）是否越出了用户空间的范围。除此之外，只有调度寄存器 7 是允许设置的，但是对其数值有些特殊的要求。

操作 PTRACE_SYSCALL 和 PTRACE_CONT 为一组，分别用来使被跟踪的子进程在下一次系统调用时暂停或继续：

==================== arch/i386/kernel/ptrace.c 289 308 ====================

[sys_ptrace()]

```
289       case PTRACE_SYSCALL: /* continue and stop at next (return from) syscall */
290       case PTRACE_CONT: { /* restart after signal. */
291            long tmp;
292
293            ret = -EIO;
294            if ((unsigned long) data > _NSIG)
295                 break;
296            if (request == PTRACE_SYSCALL)
297                 child->ptrace |= PT_TRACESYS;
298            else
299                 child->ptrace &= ~PT_TRACESYS;
300            child->exit_code = data;
301       /* make sure the single step bit is not set. */
302            tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
303            put_stack_long(child, EFL_OFFSET,tmp);
304            wake_up_process(child);
305            ret = 0;
306            break;
307       }
308
```

使子进程在下一次进入系统调用时暂停与使子进群在执行下一条指令后暂停（PTRACE_SINGLESTEP）是互斥的。所以，要将子进程的标志寄存器映象中的 TRAP_FLAG 标志清 0（302～303 行）。读者在前面已看到过 get_stack_long()的代码，而 put_stack_long()即为其逆向操作。使被跟踪进程在下一次进入系统调用时暂停是通过其 task_struct 结构中的 PF_TRACESYS 标志位起作用的。

在第 3 章讲述系统调用过程时我们有意忽略了标志位 PF_TRACESYS 的作用，现在把它补上。让我们来看看文件 arch/i386/kernel/entry.S 中的几个片断：

==================== arch/i386/kernel/entry.S 195 203 ====================

```
ENTRY(system_call)
     pushl %eax             # save orig_eax
     SAVE_ALL
     GET_CURRENT(%ebx)
     cmpl $(NR_syscalls),%eax
     jae badsys
     testb $0x02,tsk_ptrace(%ebx)     # PT_TRACESYS
     jne tracesys
     call *SYMBOL_NAME(sys_call_table)(,%eax,4)
  . . . . . .
```

==================== arch/i386/kernel/entry.S 244 254 ====================

```
tracesys:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
245         movl $-ENOSYS,EAX(%esp)
246         call SYMBOL_NAME(syscall_trace)
247         movl ORIG_EAX(%esp),%eax
248         cmpl $(NR_syscalls),%eax
249         jae tracesys_exit
250         call *SYMBOL_NAME(sys_call_table)(,%eax,4)
251         movl %eax,EAX(%esp)          # save the return value
252  tracesys_exit:
253         call SYMBOL_NAME(syscall_trace)
254         jmp ret_from_sys_call
```

在跳转到各个系统调用的处理程序之前，先要检查当前进程的 PF_TRACESYS 标志，如果为 1 就转移到 tracesys。转到 tracesys 以后，首先就是调用 syscall_trace()，其代码又回到 arch/i386/kernel/ptrace.c 中：

==================== arch/i386/kernel/ptrace.c 474 495 ====================

[system_call()>syscall_trace()]

```
asmlinkage void syscall_trace(void)
{
	if ((current->ptrace & (PT_PTRACED|PT_TRACESYS)) !=
			(PT_PTRACED|PT_TRACESYS))
		return;
	/* the 0x80 provides a way for the tracing parent to distinguish
	   between a syscall stop and SIGTRAP delivery */
	current->exit_code = SIGTRAP | ((current->ptrace & PT_TRACESYSGOOD)
					? 0x80 : 0);
	current->state = TASK_STOPPED;
	notify_parent(current, SIGCHLD);
	schedule();
	/*
	 * this isn't the same as continuing with a signal, but it will do
	 * for normal use.  strace only continues with a signal if the
	 * stopping signal is not SIGTRAP.  -brl
	 */
	if (current->exit_code) {
		send_sig(current->exit_code, current, 1);
		current->exit_code = 0;
	}
}
```

在这里，通过 notify_parent()，向父进程发送一个 SIGCHLD 信号，读者已经在第 4 章中见到过 notify_parent()的代码。然后就调用 schedule()进入暂停状态 TASK_STOPPED。当然，其父进程必定已经设置好对 SIGCHLD 信号的反应。当父进程设置了子进程的 PF_TRACESYS 标志位，然后又接收到子进程发送过来的 SIGCHLD 信号时，就知道子进程已经在系统调用的入口处陷入了暂停状态。这时候父进程就可以通过 PTRACE_PEEKUSR 等操作来收集或改变有关的数据（如调用参数）。然后，可以通过向子进程发送一个 SIGCONT 信号让它继续运行，也就是让它从 sysycall_trace()中的 schedule()返回，而回到 entry.S 中的 tracesys 处通过跳转表进入具体系统调用的代码（见 250 行）。父进程还可以通

PDF-XCHANGE Click to buy NOW! www.docu-track.com

过 PTRACE_POKEUSR 等操作将子进程的 ORIG_EAX 设置成一个大于 NT_syscalls 的值，使子进程跳过对系统调用本身的执行（见 249 行）。最后，子进程在执行完系统调用本身以后，在 tracesys_exit 处还要再调用一次 syscall_trace()，让父进程有个机会来收集子进程在执行完系统调用后的结果（如返回值或出错代码）。这样，父进程就可以监视子进程的所有系统调用，甚至还能向子进程“伪造”对系统调用的执行，把子进程的系统调用“重定向”到父进程的用户空间程序中。

回到 arch/i386/kernel/ptrace.c 中函数 sys_ptrace()的代码继续往下看，下面比较简单一些了：

==================== arch/i386/kernel/ptrace.c 309 327 ====================

[sys_ptrace()]

```
/*
 * make the child exit.  Best I can do is send it a sigkill.
 * perhaps it should be put in the status that it wants to
 * exit.
 */
     case PTRACE_KILL: {
          long tmp;

          ret = 0;
          if (child->state == TASK_ZOMBIE) /* already dead */
               break;
          child->exit_code = SIGKILL;
          /* make sure the single step bit is not set. */
          tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
          put_stack_long(child, EFL_OFFSET, tmp);
          wake_up_process(child);
          break;
     }
```

PTRACE_KILL 操作使子进程退出运行。除 PTRACE_ATTACH 以外，其它的操作一般都要求目标进程处于暂停状态（只有这样，目标进程的内存和寄存器映象才是静态的），只有 PTRACE_KILL 是个例外（见前面的 200～201 行所作的检查）。函数 wake_up_process()将目标进程的状态改成 TASK_RUNNING，而不问其原来是什么状态。如果子进程处于 PF_TRACESYS 状态，则当子进程下一次进行系统调用而在内核中进入 syscall_trace()以后，会向其自身发送一个 SIGKILL 信号（见上面的 479～481 行）。继续往下看：

==================== arch/i386/kernel/ptrace.c 328 347 ====================

[sys_ptrace()]

```
     case PTRACE_SINGLESTEP: {  /* set the trap flag. */
          long tmp;

          ret = -EIO;
          if ((unsigned long) data > _NSIG)
               break;
          child->ptrace &= ~PT_TRACESYS;
          if ((child->ptrace & PT_DTRACE) == 0) {
               /* Spurious delayed TF traps may occur */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                child->ptrace |= PT_DTRACE;
            }
            tmp = get_stack_long(child, EFL_OFFSET) | TRAP_FLAG;
            put_stack_long(child, EFL_OFFSET, tmp);
            child->exit_code = data;
            /* give it a chance to run. */
            wake_up_process(child);
            ret = 0;
            break;
        }

```

除通过前述的调试寄存器可以让寄存器在特定的条件下进入 1 号陷阱 debug 外，i386 CPU 还提供了单步执行的手段。只要在处理器的标志寄存器中将 TRAP_FLAG 标志位设成 1，处理器就会在每执行完一条机器指令以后就进入 debug 陷阱而到达一个断点。这样，跟踪进程就可以像对待子进程的系统调用一样，在子进程每执行完一条指令后就来观察执行的结果。不过，对指令的跟踪与对系统调用的跟踪是互斥的，所以要将子进程的 PF_TRACESYS 标志清 0。注意 PF_TRACESYS 标志与 TRAP_FLAG 是两码事。前者是一个软件标志，是进程的 task_struct 结构内部 flags 中的一位，这完全是供软件使用的，而与处理器的硬件没有直接的联系。相比之下，TRAP_FLAG 是个硬件标志，它是处理器中的“标志寄存器”EFL 的一位，直接影响着处理器的行为。每当调度一个进程进入运行时，就会在返回用户空间前夕将其标志寄存器映象装入 CPU 的标志寄存器 EFL，所以可以实现对该进程的单步跟踪，而并不影响其它进程或者在系统空间的运行。但是，直接用 TRAP_FLAG 标志位来代表 ptrace()机制的单步跟踪状态是不可靠的。这是因为应用软件也可以改变处理器的标志寄存器，从而造成混淆。所以，ptrace()同时还定义了一个用于单步执行的软件标志 PT_DTRACE，在通过 ptrace()的 PTRACE_SINGLESTEP 来开始单步跟踪时就将这个软件标志也设成 1。其余的操作就比较简单了。结合前面一些操作的代码，读者自行阅读应该不会有困难（arch/i386/kernel/ptrace.c）。

==================== arch/i386/kernel/ptrace.c 348 463 ====================

[sys_ptrace()]

```
        case PTRACE_DETACH: { /* detach a process that was attached. */
            long tmp;

            ret = -EIO;
            if ((unsigned long) data > _NSIG)
                break;
            child->ptrace = 0;
            child->exit_code = data;
            write_lock_irq(&tasklist_lock);
            REMOVE_LINKS(child);
            child->p_pptr = child->p_opptr;
            SET_LINKS(child);
            write_unlock_irq(&tasklist_lock);
            /* make sure the single step bit is not set. */
            tmp = get_stack_long(child, EFL_OFFSET) & ~TRAP_FLAG;
            put_stack_long(child, EFL_OFFSET, tmp);
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        wake_up_process(child);
        ret = 0;
        break;
    }

    case PTRACE_GETREGS: { /* Get all gp regs from the child. */
        if (!access_ok(VERIFY_WRITE, (unsigned *)data, 17*sizeof(long))) {
            ret = -EIO;
            break;
        }
        for ( i = 0; i < 17*sizeof(long); i += sizeof(long) ) {
            __put_user(getreg(child, i),(unsigned long *) data);
            data += sizeof(long);
        }
        ret = 0;
        break;
    }

    case PTRACE_SETREGS: { /* Set all gp regs in the child. */
        unsigned long tmp;
        if (!access_ok(VERIFY_READ, (unsigned *)data, 17*sizeof(long))) {
            ret = -EIO;
            break;
        }
        for ( i = 0; i < 17*sizeof(long); i += sizeof(long) ) {
            __get_user(tmp, (unsigned long *) data);
            putreg(child, i, tmp);
            data += sizeof(long);
        }
        ret = 0;
        break;
    }

    case PTRACE_GETFPREGS: { /* Get the child FPU state. */
        if (!access_ok(VERIFY_WRITE, (unsigned *)data,
                  sizeof(struct user_i387_struct))) {
            ret = -EIO;
            break;
        }
        ret = 0;
        if ( !child->used_math ) {
            /* Simulate an empty FPU. */
            set_fpu_cwd(child, 0x037f);
            set_fpu_swd(child, 0x0000);
            set_fpu_twd(child, 0xffff);
        }
        get_fpregs((struct user_i387_struct *)data, child);
        break;
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
case PTRACE_SETFPREGS: { /* Set the child FPU state. */
    if (!access_ok(VERIFY_READ, (unsigned *)data,
                sizeof(struct user_i387_struct))) {
        ret = -EIO;
        break;
    }
    child->used_math = 1;
    set_fpregs(child, (struct user_i387_struct *)data);
    ret = 0;
    break;
}

case PTRACE_GETFPXREGS: { /* Get the child extended FPU state. */
    if (!access_ok(VERIFY_WRITE, (unsigned *)data,
                sizeof(struct user_fxsr_struct))) {
        ret = -EIO;
        break;
    }
    if ( !child->used_math ) {
        /* Simulate an empty FPU. */
        set_fpu_cwd(child, 0x037f);
        set_fpu_swd(child, 0x0000);
        set_fpu_twd(child, 0xffff);
        set_fpu_mxcsr(child, 0x1f80);
    }
    ret = get_fpxregs((struct user_fxsr_struct *)data, child);
    break;
}

case PTRACE_SETFPXREGS: { /* Set the child extended FPU state. */
    if (!access_ok(VERIFY_READ, (unsigned *)data,
                sizeof(struct user_fxsr_struct))) {
        ret = -EIO;
        break;
    }
    child->used_math = 1;
    ret = set_fpxregs(child, (struct user_fxsr_struct *)data);
    break;
}

case PTRACE_SETOPTIONS: {
    if (data & PTRACE_O_TRACESYSGOOD)
        child->ptrace |= PT_TRACESYSGOOD;
    else
        child->ptrace &= ~PT_TRACESYSGOOD;
    ret = 0;
    break;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

	default:
		ret = -EIO;
		break;
	}
out_tsk:
	free_task_struct(child);
out:
	unlock_kernel();
	return ret;
}
```

如前所述，ptrace()在实际运用中并不是用作进程间通讯的手段，而是作为程序调试和维护的手段。作为调试手段，其各方面的作用在早期的 Unix 系统中是无可替代的。不过，随着 Unix（以及 Linux）的发展，出现了/proc 目录下的特殊文件（见“文件系统”一章中的有关内容），使用户可以通过这些特殊文件来读 / 写一个进程的内存空间和其它信息，而且往往更为方便，形式上也更为划一。所以，近年来像 gdb 一类的调试工具已经倾向于更多地使用这些特殊文件（严格说来这些特殊文件当然也可用于进程间通讯，只不过人们已经有了更好的进程间通讯手段，因而不会这样去用而已）。但是，尽管如此，/proc 特殊文件还是不能完全取代 ptrace()的作用。例如，ptrace()有个无可替代的作用，那就是可以通过跟踪应用程序所作的系统调用来监视其运行。我们知道，Linux 内核的源代码是公开的，可是应用程序的源代码却一般都不公开。拿到一个应用程序以后，如果想要知道它究竟在干些什么，最好的办法就是监视它都作了些什么系统调用，调用时的参数都是些什么，返回值又是什么。这时就要用到 ptrace()了。为了这个目的，Linux 专门提供了一个工具，即 shell 实用程序 strace。读者不妨先体验一下 strace 的使用，然后，想想它是怎样实现的？

% strace ceho hello

6.6 报文传递

从本节开始，我们用三节的篇幅集中介绍 Linux 内核对 Unix 系统 V 进程间通信机制的继承和实现。

早期 Unix 系统的进程间通信（IPC）机制主要有两种，就是管道和信号。后来，针对普通管道只能在“近亲”进程之间建立的缺点，又有了命名管道。但是，对于一个现代的操作系统以及日益发展的各种应用来说，这些机制虽然很重要，但也确实存在明显的不足。首先，信号所能载送的信息量很小，单独使用时不适合信息量要求比较大的场合。而管道，即使是命名管道，虽然可用于信息量较大的场合，但是对于不同的应用而言还是有许多缺点，主要有：

- 所载送的信息是无格式的字节流。设想如果一个进程要发送两段文字给另一个进程，每段文字都是一个小小的字符文件，那么对接收方进程而言，这两段文字连成了一片，怎样知道这两段文字的分界在哪儿呢？一个管道（命名的或者无名的）就好像是一条通信线路，在这条通信线路上要有一些起码的、低层的“通信规程”，例如报文的划分，才能满足较高层规程的要求。从这个角度来说，无格式字节流只是一种最原始的通信手段。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

- 由于所载送的是无格式字节流，就缺乏一些控制手段，如报文的优先级别等。
- 管道机制的缓冲区大小是有限的、静态的。当发送者写满了缓冲区而接收者没有来得及从缓冲区读走，发送者就只好停下来睡眠，这就强化了管道机制的同步性要求。虽然这种同步性往往本来就是应用程序所需要的，但在某些应用中、某些场合下却成为一个缺陷。固然，可以通过使用 O_NONBLOCK 标志让发送者在缓冲区一满就返回（而不至于进入睡眠），但那样会增加应用程序的复杂性，还会进一步降低效率。再说，在管道机制中也无法让发送者预先知道缓冲区中还有多少可写空间。
- 从运行效率看，管道机制的开销也不小，尤其是当发送的信息量比较小时，平均每个字节所耗费的代价就相当高了。
- 每个管道都要占用一个打开文件号，一般的应用中这还不至于成为问题，但在某些特殊的应用中是有可能会成为问题的。

上述的多数缺陷都可以在应用软件中采取一些措施加以克服或减轻，但是那样只会使应用软件更复杂、效率更低。而操作系统的作用和目的本来就在于使应用软件更简单、更安全、更高效。这样，针对各种应用的要求，就提出了改进早期 Unix IPC 机制的任务。另一方面，当时在操作系统以及一些有关领域的理论研究也有了较大的发展而且日趋成熟，概括出了对 IPC 的几种抽象，包括报文（message）传递、共享内存以及进程同步。其中进程同步又包括了信号量（semaphore）、互斥量（mutex）以及“约会”（rendezvous）等形式。在这样的历史条件下，AT&T 在其 Unix 系统 V 版本中增加了报文传递、共享内存以及信号量三种 IPC 机制。这三种机制合在一起统称为“系统 V 进程间通信机制”。与此同时，BSD 也在早期 Unix IPC 机制的基础上作了扩充，形成了基于 socket 的 IPC 机制，使得 IPC 成为系统 V 与 BSD 版本之间的主要区别之一。而 Linux 则兼收并蓄，把两者都继承了下来。我们将在下一章再介绍 BSD 所扩充的 IPC 机制，而本节以及后面两节则集中于 AT&T 的系统 V IPC 机制在 Linux 内核中的具体实现。

Linux 内核为系统 V IPC 提供了一个统一的系统调用 ipc()，也许应称为 sysv_ipc()。其应用程序设计界面（API）为：

int ipc(unsigned int call, int first, int second, int third, void *ptr, int forth);

其中第一个参数 call 为具体的操作码，定义于 include/asm-i386/ipc.h 中：

==================== include/asm-i386/ipc.h 14 24 ====================

```
#define SEMOP        1
#define SEMGET           2
#define SEMCTL           3
#define MSGSND          11
#define MSGRCV          12
#define MSGGET          13
#define MSGCTL          14
#define SHMAT       21
#define SHMDT       22
#define SHMGET          23
#define SHMCTL          24
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

操作码中凡由“SEM”开头的都是为信号量而设，由“MSG”开头的都是为报文传递而设，而由“SHM”开头的则都是为共享内存区而设。其余参数的使用则因具体操作的不同而异。不过，为便于使用，在 C 语言函数库中分别提供了如 semget()、msgget()、msgsnd()等等库函数，这些库函数把用户程序对它们的调用转换成统一的系统调用 ipc()，却使用户感到好像内核提供了这么一些不同的系统调用一样。

内核中的入口为 sys_ipc()，其代码在 arch/i386/kernel/sys_i386.c 中：

==================== arch/i386/kernel/sys_i386.c 127 205 ====================

```
/*
 * sys_ipc() is the de-multiplexer for the SysV IPC calls..
 *
 * This is really horribly ugly.
 */
asmlinkage int sys_ipc (uint call, int first, int second,
              int third, void *ptr, long fifth)
{
     int version, ret;

     version = call >> 16; /* hack for backward compatibility */
     call &= 0xffff;

     switch (call) {
     case SEMOP:
          return sys_semop (first, (struct sembuf *)ptr, second);
     case SEMGET:
          return sys_semget (first, second, third);
     case SEMCTL: {
          union semun fourth;
          if (!ptr)
               return -EINVAL;
          if (get_user(fourth.__pad, (void **) ptr))
               return -EFAULT;
          return sys_semctl (first, second, third, fourth);
     }

     case MSGSND:
          return sys_msgsnd (first, (struct msgbuf *) ptr,
                       second, third);
     case MSGRCV:
          switch (version) {
          case 0: {
               struct ipc_kludge tmp;
               if (!ptr)
                    return -EINVAL;

               if (copy_from_user(&tmp,
                             (struct ipc_kludge *) ptr,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        sizeof (tmp)))
                return -EFAULT;
            return sys_msgrcv (first, tmp.msgp, second,
                        tmp.msgtyp, third);
        }
        default:
            return sys_msgrcv (first,
                        (struct msgbuf *) ptr,
                        second, fifth, third);
        }
    case MSGGET:
        return sys_msgget ((key_t) first, second);
    case MSGCTL:
        return sys_msgctl (first, second, (struct msqid_ds *) ptr);

    case SHMAT:
        switch (version) {
        default: {
            ulong raddr;
            ret = sys_shmat (first, (char *) ptr, second, &raddr);
            if (ret)
                return ret;
            return put_user (raddr, (ulong *) third);
        }
        case 1:  /* iBCS2 emulator entry point */
            if (!segment_eq(get_fs(), get_ds()))
                return -EINVAL;
            return sys_shmat (first, (char *) ptr, second, (ulong *) third);
        }
    case SHMDT:
        return sys_shmdt ((char *)ptr);
    case SHMGET:
        return sys_shmget (first, second, third);
    case SHMCTL:
        return sys_shmctl (first, second,
                    (struct shmid_ds *) ptr);
    default:
        return -EINVAL;
    }
}
```

函数 sys_ipc 的结构很简单：根据调用参数操作码的不同，分别处理三种进程间通信机制的 11 项不同的操作。我们分三节介绍相关的源代码，本节先介绍报文传递。

每一个进程都可以通过库函数调用 msgget()（即通过操作码为 MSGGET 的 ipc()系统调用，下同）建立报文队列。有的系统中把这样的队列称为“信箱”（mail box）。报文队列不是通过文件名，而是通过一个“键值”（key）加以标识的。一旦建立以后，其它进程就可以使用相同的键值通过 msgget()取得对已建立报文队列的访问途径。然后，发送报文的进程就可以通过 msgsnd()发送报文到指定的队列中，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

而接收进程则可以通过 msgrcv()从指定的队列中接收报文。从概念上说，这类似于命名管道，但报文队列所传递的不再是完全无结构的字节流了，每个报文都有一定的低层结构而互相区分。此外，还可以通过 msgctl()调用对指定的报文队列进行一些控制（包括撤消该队列）。由于报文队列并不纳入文件系统的范畴，所以并不占用打开文件号。内核中实现报文传递机制的代码基本上都在文件 ipc/msg.c 中。

6.6.1 库函数 msgget()——创建报文队列

先来看报文队列的建立和取得（ipc/msg.c）：

==================== ipc/msg.c 303 330 ====================

[sys_ipc()>sys_msgget()]

```
asmlinkage long sys_msgget (key_t key, int msgflg)
{
	int id, ret = -EPERM;
	struct msg_queue *msq;

	down(&msg_ids.sem);
	if (key == IPC_PRIVATE)
		ret = newque(key, msgflg);
	else if ((id = ipc_findkey(&msg_ids, key)) == -1) { /* key not used */
		if (!(msgflg & IPC_CREAT))
			ret = -ENOENT;
		else
			ret = newque(key, msgflg);
	} else if (msgflg & IPC_CREAT && msgflg & IPC_EXCL) {
		ret = -EEXIST;
	} else {
		msq = msg_lock(id);
		if(msq==NULL)
			BUG();
		if (ipcperms(&msq->q_perm, msgflg))
			ret = -EACCES;
		else
			ret = msg_buildid(id, msq->q_perm.seq);
		msg_unlock(id);
	}
	up(&msg_ids.sem);
	return ret;
}
```

操作码 MSGGET，从而是 sys_msgget()可用于两个不同的目的：一是用一个给定的键值创建一个报文队列；二是给定一个键值，找到已经建立的报文队列。当调用参数 msgflg 中的 IPC_CREATE 标志为 1 时表示创建，为 0 时则为寻找，二者均返回报文队列的标识号。内核中有个全局的数据结构 msg_ids，专门用来管理报文队列（ipc/msg.c）。

==================== ipc/msg.c 92 92 ====================

```
static struct ipc_ids msg_ids;
```

其中类型 ipc_ids 在 ipc/util.h 中定义：

==================== ipc/util.h 15 24 ====================

```
struct ipc_ids {
      int size;
      int in_use;
      int max_id;
      unsigned short seq;
      unsigned short seq_max;
      struct semaphore sem;
      spinlock_t ary;
      struct ipc_id* entries;
};
```

结构中的指针 entries 指向一个结构数组，其类型定义为（ipc/util.h）：

==================== ipc/util.h 26 28 ====================

```
struct ipc_id {
      struct kern_ipc_perm* p;
};
```

数组中的元素都是指向 kern_ipc_perm 数据结构的指针，而 kern_ipc_perm 数据结构则是在 include/linux/ipc.h 中定义的：

==================== include/linux/ipc.h 56 66 ====================

```
/* used by in-kernel data structures */
struct kern_ipc_perm
{
      key_t          key;
      uid_t          uid;
      gid_t          gid;
      uid_t          cuid;
      gid_t          cgid;
      mode_t         mode;
      unsigned long seq;
};
```

数据结构 msg_ids 是全局的，显然必须置于内核中“信号量”机制（见第 4 章）的保护之下。

键的类型为 key_t，实际上是个整数。前面讲过，报文队列以键值而不是文件名来标识，所以每个报文队列的键值必须是惟一的。不过，键值 0，也就是 IPC_PRIVATE，是一种特殊情况。每个进程都可以用键值 0 建立一个专供其私用（自己发送自己接收）的报文队列。所以这个特殊键值并不是惟一的。但是，正由于这种队列是私用的，所以不存在要通过键值找到一个队列的问题。

正因为这样，当键值为 IPC_PRIVATE 时（309 行），就无条件地调用 newque()建立一个报文队列（310 行），否则就要先通过 ipc_findkey()找一下相应的报文队列是否已经存在。

函数 newque()的代码在同一文件 ipc/msg.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== ipc/msg.c 117 144 ====================

[sys_ipc()>sys_msgget()>newque()]

```
static int newque (key_t key, int msgflg)
{
	int id;
	struct msg_queue *msq;

	msq  = (struct msg_queue *) kmalloc (sizeof (*msq), GFP_KERNEL);
	if (!msq)
		return -ENOMEM;
	id = ipc_addid(&msg_ids, &msq->q_perm, msg_ctlmni);
	if(id == -1) {
		kfree(msq);
		return -ENOSPC;
	}
	msq->q_perm.mode = (msgflg & S_IRWXUGO);
	msq->q_perm.key = key;

	msq->q_stime = msq->q_rtime = 0;
	msq->q_ctime = CURRENT_TIME;
	msq->q_cbytes = msq->q_qnum = 0;
	msq->q_qbytes = msg_ctlmnb;
	msq->q_lspid = msq->q_lrpid = 0;
	INIT_LIST_HEAD(&msq->q_messages);
	INIT_LIST_HEAD(&msq->q_receivers);
	INIT_LIST_HEAD(&msq->q_senders);
	msg_unlock(id);

	return msg_buildid(id,msq->q_perm.seq);
}
```

每个报文队列都有个队列头，那就是 msg_queue 数据结构，定义于 ipc/msg.c 中：

==================== ipc/msg.c 67 82 ====================

```
/* one msq_queue structure for each present queue on the system */
struct msg_queue {
	struct kern_ipc_perm q_perm;
	time_t q_stime;			/* last msgsnd time */
	time_t q_rtime;			/* last msgrcv time */
	time_t q_ctime;			/* last change time */
	unsigned long q_cbytes;		/* current number of bytes on queue */
	unsigned long q_qnum;		/* number of messages in queue */
	unsigned long q_qbytes;		/* max number of bytes on queue */
	pid_t q_lspid;			/* pid of last msgsnd */
	pid_t q_lrpid;			/* last receive pid */

	struct list_head q_messages;
	struct list_head q_receivers;
	struct list_head q_senders;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
};
```

反之，每个 msg_queue 数据结构也惟一地对应着一个报文队列。这些数据结构以及结构之间的关系可以总结如下：

- 全局量 ipc_ids 数据结构 msg_ids 是系统中所有报文队列的“总根”。
- 数据结构 msg_ids 中的指针 entries 指向一个 ipc_id 结构数组，数组中的每个元素都是 ipc_id 数据结构，结构中有个指针 p，指向一个 kern_ipc_perm 数据结构。
- 由于 kern_ipc_perm 数据结构是报文队列头 msg_queue 数据结构内部的第一个成分，上述数组元素中的指针 p 实际上指向一个报文队列，数组的大小决定了已经或可以建立的报文队列数量。

每个已建立的报文队列由一个标识号来代表，与打开文件号相似。但是，打开文件号只局限于每个进程局部，而报文队列的标识号却是全局的，所以必须保证在全局范围中的惟一性。标识号由 ipc_addid()分配，其代码在 ipc/util.c 中：

==================== ipc/util.c 134 171 ====================

[sys_ipc()>sys_msgget()>newque()>ipc_addid()]

```
/**
 *  ipc_addid     -    add an IPC identifier
 *  @ids: IPC identifier set
 *  @new: new IPC permission set
 *  @size: new size limit for the id array
 *
 *  Add an entry 'new' to the IPC arrays. The permissions object is
 *  initialised and the first free entry is set up and the id assigned
 *  is returned. The list is returned in a locked state on success.
 *  On failure the list is not locked and -1 is returned.
 */

int ipc_addid(struct ipc_ids* ids, struct kern_ipc_perm* new, int size)
{
     int id;

     size = grow_ary(ids,size);
     for (id = 0; id < size; id++) {
          if(ids->entries[id].p == NULL)
               goto found;
     }
     return -1;
found:
     ids->in_use++;
     if (id > ids->max_id)
          ids->max_id = id;

     new->cuid = new->uid = current->euid;
     new->gid = new->cgid = current->egid;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
164         new->seq = ids->seq++;
165         if(ids->seq > ids->seq_max)
166              ids->seq = 0;
167
168         spin_lock(&ids->ary);
169         ids->entries[id].p = new;
170         return id;
171    }
```

如果分配标识号成功，就要将代表报文队列的报文队列头与 ipc_ids 结构 msg_ids 挂上钩。如前所述，该结构中的指针 entries 指向以标识号为下标的 ipc_id 结构数组，而 ipc_id 结构的内容只是一个指针，指向一个 kern_ipc_perm 结构。同时，每个报文队列头结构中的第一个成分就是一个 kern_ipc_perm 数据结构，其起始地址与整个 msg_queue 结构的起始地址相同。所谓将某个报文队列头与 msg_ids 挂上钩，就是把特定 msg_queue 结构中 kern_ipc_perm 数据结构的起始地址根据标识号填入相应 ipc_id 结构中（见 169 行），并返回该标识号。

然而，既然是数组，就有个大小，ipc_ids 结构中的字段 size 就记录着它的大小。可是这个大小怎样确定呢？再说，再大的数组也有可能会用完，那时候又怎么办？显然，最好是能根据实际的需要加以调整，这就是 ipc_addid()的代码中调用 grow_ary()的目的（ipc/util.c）：

==================== ipc/util.c 105 132 ====================

[sys_ipc()>sys_msgget()>newque()>ipc_addid()>grow_ary()]

```
105    static int grow_ary(struct ipc_ids* ids, int newsize)
106    {
107         struct ipc_id* new;
108         struct ipc_id* old;
109         int i;
110
111         if(newsize > IPCMNI)
112              newsize = IPCMNI;
113         if(newsize <= ids->size)
114              return newsize;
115
116         new = ipc_alloc(sizeof(struct ipc_id)*newsize);
117         if(new == NULL)
118              return ids->size;
119         memcpy(new, ids->entries, sizeof(struct ipc_id)*ids->size);
120         for(i=ids->size;i<newsize;i++) {
121              new[i].p = NULL;
122         }
123         spin_lock(&ids->ary);
124
125         old = ids->entries;
126         ids->entries = new;
127         i = ids->size;
128         ids->size = newsize;
129         spin_unlock(&ids->ary);
130         ipc_free(old, sizeof(struct ipc_id)*i);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
131         return ids->size;
132     }
```

参数 newsize 表示新的数组大小，如果其数值大于数组当前的大小就另外分配一块空间来取代原有的数组。不过，数组的大小只会扩展而不会缩小（见 113 行）。另一方面，数组的扩展也有个上限 IPCMNI，该常数在 include/linux/ipc.h 中定义为 32768。

```
#define IPCMNI 32768  /* <= MAX_INT limit for ipc arrays (including sysctl changes) */
```

在 newque()中调用 ipc_addid()时将一个全局量 msg_ctlmni 作为参数传了下来，就是这里的 newsize。这个全局量的初值为 MSGMNI，而 MSGMNI 在 msg.h 中定义为 16。

最后，newque()还要将这个标识号转换成一个一体化的标识号。代码中的 msg_buildid()是个宏定义（ipc/msg.c）：

==================== ipc/msg.c 99 100 ====================

```
#define msg_buildid(id, seq) \
        ipc_buildid(&msg_ids, id, seq)
```

而 ipc_buildid()的定义则在 ipc/util.h 中：

==================== ipc/util.h 87 90 ====================

```
extern inline int ipc_buildid(struct ipc_ids* ids, int id, int seq)
{
        return SEQ_MULTIPLIER*seq + id;
}
```

为什么要作这样的转换呢？从 ipc_addid()的代码中可以看出，由它分配的标识号实际上是 msg_ids 结构数组中的数组下标。这个下标是重复使用的。如果一个进程建立了一个报文队列，然后又撤消了，而后来又有另一个进程要建立一个报文队列，就有可能又将同一个下标分配给这个新的队列。这样，虽然这种标识号的使用在任何一个特定的时间点上是惟一的，但是如果观察一个合理长的时间跨度就不一定是惟一的了。为了克服这个问题，在 msg_ids 中设立了一个序号 seq；每分配使用一个标识号时就递增这个序号（见 ipc_addid()中的第 164 行），并且将这个序号与下标编码在一起形成一个一体化的标识号。这么一来，即使在一段时间以后下标又重复了，但由于序号在递增，所以一体化的标识号在相当长的一段时期里都能保证惟一性。

还要注意，键值与标识号是两码事。用文件系统打个比方，则键值类似于文件名，而标识号类似于打开文件号。

回到 sys_msgget()中。如果键值不是 IPC_PRIVATE，那就先寻找以给定键值建立的报文队列是否已经存在。函数 ipc_findkey 的代码在 ipc/util.c 中：

==================== ipc/util.c 82 101 ====================

[sys_msgget()>ipc_findkey()]

```
/**
 *  ipc_findkey   -   find a key in an ipc identifier set
 *  @ids: Identifier set
 *  @key: The key to find
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   Returns the identifier if found or -1 if not.
 */

int ipc_findkey(struct ipc_ids* ids, key_t key)
{
     int id;
     struct kern_ipc_perm* p;

     for (id = 0; id <= ids->max_id; id++) {
          p = ids->entries[id].p;
          if(p==NULL)
               continue;
          if (key == p->key)
               return id;
     }
```

寻找的结果对于调用 sys_msgget()的不同目的（创建或寻找）有着不同的意义，因此要分不同的情形来处理：①如果找不到，那么对于寻找的目的来说是一次失败，而对于创建的目的来说，却是好事，可以进而调用 newque()了。②如果找到了，那么对于独占性的创建（IPC_EXCL 标志为 1），这是一次失败，而对于寻找或可共享的创建来说却是好事。不过，一个已经建立的报文队列并不是谁都可以来使用或共享的。一般情况下，只有与创建者属于同一用户并且同一组或者属于超级用户的进程才有资格来使用或共享。所以在 sys_msgget()中，要调用 ipcperms()先检查一下访问权限是否相符。最后，同样还要通过 msg_buildid()将实际上是数组下标的标识号转换成一体化的标识号。

函数 ipcperms()的代码在 ipc/util.c 中：

==================== ipc/util.c 242 267 ====================

[sys_msgget()>ipcperms()]

```
/**
 *   ipcperms  -   check IPC permissions
 *   @ipcp: IPC permission set
 *   @flag: desired permission set.
 *
 *   Check user, group, other permissions for access
 *   to ipc resources. return 0 if allowed
 */

int ipcperms (struct kern_ipc_perm *ipcp, short flag)
{    /* flag will most probably be 0 or S_...UGO from <linux/stat.h> */
     int requested_mode, granted_mode;

     requested_mode = (flag >> 6) | (flag >> 3) | flag;
     granted_mode = ipcp->mode;
     if (current->euid == ipcp->cuid || current->euid == ipcp->uid)
          granted_mode >>= 6;
     else if (in_group_p(ipcp->cgid) || in_group_p(ipcp->gid))
          granted_mode >>= 3;
     /* is there some bit set in requested_mode but not in granted_mode? */
```

```
    if ((requested_mode & ~granted_mode & 0007) &&
        !capable(CAP_IPC_OWNER))
         return -1;

    return 0;
}
```

可见，报文队列访问权限的管理与文件系统访问权限的管理相似，读者可参考第 5 章中有关的内容。

6.6.2 库函数 msgsnd()——报文发送

参与通信的双方在通过 msgget()创建了一个报文队列或取得了该队列的标识号以后，就可以向该队列发送或接收报文了。先来看报文的发送，这个函数比较大，所以我们还是分段来阅读（ipc/msg.c）。

==================== ipc/msg.c 626 650 ====================

```
asmlinkage long sys_msgsnd (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)
{
    struct msg_queue *msq;
    struct msg_msg *msg;
    long mtype;
    int err;

    if (msgsz > msg_ctlmax || (long) msgsz < 0 || msqid < 0)
         return -EINVAL;
    if (get_user(mtype, &msgp->mtype))
         return -EFAULT;
    if (mtype < 1)
         return -EINVAL;

    msg = load_msg(msgp->mtext, msgsz);
    if(IS_ERR(msg))
         return PTR_ERR(msg);

    msg->m_type = mtype;
    msg->m_ts = msgsz;

    msq = msg_lock(msqid);
    err=-EINVAL;
    if(msq==NULL)
         goto out_free;
```

首先是对参数作一些检查并将报文从用户空间复制过来，其中 msgp 是指向一个 msgbuf 结构的指针，这个数据结构是在 include/linux/msg.h 中定义的：

==================== include/linux/msg.h 34 38 ====================

```
/* message buffer for msgsnd and msgrcv calls */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
35  struct msgbuf {
36      long mtype;         /* type of message */
37      char mtext[1];      /* message text */
38  };
```

虽然这个指针已经作为系统调用的参数传了过来，数据结构本身却还在用户空间，所以分别通过 get_user()和 load_msg()从用户空间复制到系统空间。其中 load_msg()还要在系统空间为此分配缓冲区。系统空间中使用的 msg_msg 结构与用户空间中使用的 msgbuf 结构不同（ipc/msg.c）：

==================== ipc/msg.c 55 62 ====================

```
55  /* one msg_msg structure for each message */
56  struct msg_msg {
57      struct list_head m_list;
58      long  m_type;
59      int m_ts;           /* message text size */
60      struct msg_msgseg* next;
61      /* the actual message follows immediately */
62  };
```

当报文本身的大小加上这个数据结构的大小仍小于一个页面时，msg_msg 结构与报文本身在同一页面中，而报文本身紧跟在 msg_msg 数据结构后面。否则，当报文本身和 msg_msg 结构不能容纳在同一页面中时，则要将报文分段，然后将分布于不同页面中的报文段链接起来。此时除第一个页面的开头仍是 msg_msg 结构以外，其它各个页面的开头都是一个 msg_msgseg 结构，而每个报文段的大小则为页面大小减去 msg_msgseg 结构的大小（ipc/msg.c）：

==================== ipc/msg.c 51 54 ====================

```
51  struct msg_msgseg {
52      struct msg_msgseg* next;
53      /* the next part of the message follows immediately */
54  };
```

函数 load_msg()的代码也在 ipc/msg.c 中，我们把它留给读者自己阅读。

==================== ipc/msg.c 158 208 ====================

[sys_msgsnd()>load_msg()]

```
158 static struct msg_msg* load_msg(void* src, int len)
159 {
160     struct msg_msg* msg;
161     struct msg_msgseg** pseg;
162     int err;
163     int alen;
164 
165     alen = len;
166     if(alen > DATALEN_MSG)
167         alen = DATALEN_MSG;
168 
169     msg = (struct msg_msg *) kmalloc (sizeof(*msg) + alen, GFP_KERNEL);
170     if(msg==NULL)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
171             return ERR_PTR(-ENOMEM);
172
173        msg->next = NULL;
174
175        if (copy_from_user(msg+1, src, alen)) {
176             err = -EFAULT;
177             goto out_err;
178        }
179
180        len -= alen;
181        src = ((char*)src)+alen;
182        pseg = &msg->next;
183        while(len > 0) {
184             struct msg_msgseg* seg;
185             alen = len;
186             if(alen > DATALEN_SEG)
187                  alen = DATALEN_SEG;
188             seg = (struct msg_msgseg *) kmalloc (sizeof(*seg) + alen, GFP_KERNEL);
189             if(seg==NULL) {
190                  err=-ENOMEM;
191                  goto out_err;
192             }
193             *pseg = seg;
194             seg->next = NULL;
195             if(copy_from_user (seg+1, src, alen)) {
196                  err = -EFAULT;
197                  goto out_err;
198             }
199             pseg = &seg->next;
200             len -= alen;
201             src = ((char*)src)+alen;
202        }
203        return msg;
204
205   out_err:
206        free_msg(msg);
207        return ERR_PTR(err);
208   }
```

回到 sys_msgsnd()的代码中。647 行函数 msg_lock()的作用是根据给定的标识号找到相应的报文队列，并将其数据结构上锁。

至此，所需的准备工作基本完成了，我们继续在 ipc/msg.c 中往下看：

==================== ipc/msg.c 651 684 ====================

[sys_msgsnd()]

```
651   retry:
652        err= -EIDRM;
653        if (msg_checkid(msq,msqid))
654             goto out_unlock_free;
```

```

    err=-EACCES;
    if (ipcperms(&msq->q_perm, S_IWUGO))
        goto out_unlock_free;

    if(msgsz + msq->q_cbytes > msq->q_qbytes ||
        1 + msq->q_qnum > msq->q_qbytes) {
        struct msg_sender s;

        if(msgflg&IPC_NOWAIT) {
            err=-EAGAIN;
            goto out_unlock_free;
        }
        ss_add(msq, &s);
        msg_unlock(msqid);
        schedule();
        current->state= TASK_RUNNING;

        msq = msg_lock(msqid);
        err = -EIDRM;
        if(msq==NULL)
            goto out_free;
        ss_del(&s);

        if (signal_pending(current)) {
            err=-EINTR;
            goto out_unlock_free;
        }
        goto retry;
    }
```

在用户界面上使用的队列标识号是经过编码的一体化标识号，这里还要再检验一下。可是，既然这个队列是通过 msg_lock()找到的，为什么还要再检验呢？看一下 msg_lock()和 msg_checkid()的代码就清楚了（ipc/msg.c）：

==================== ipc/msg.c 94 94 ====================

```
#define msg_lock(id)   ((struct msg_queue*)ipc_lock(&msg_ids,id))
. . . . . .
```

==================== ipc/msg.c 97 98 ====================

```
#define msg_checkid(msq, msgid) \
    ipc_checkid(&msg_ids,&msq->q_perm,msgid)
```

两个 inline 函数的定义都在 ipc/util.h 中：

==================== ipc/util.h 68 80 ====================

[sys_msgsnd()>msg_lock()>ipc_lock()]

```
extern inline struct kern_ipc_perm* ipc_lock(struct ipc_ids* ids, int id)
{
    struct kern_ipc_perm* out;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int lid = id % SEQ_MULTIPLIER;
    if(lid > ids->size)
        return NULL;

    spin_lock(&ids->ary);
    out = ids->entries[lid].p;
    if(out==NULL)
        spin_unlock(&ids->ary);
    return out;
}
```

==================== ipc/util.h 92 97 ====================

[sys_msgsnd()>msg_checkid()>ipc_checkid()]

```
extern inline int ipc_checkid(struct ipc_ids* ids, struct kern_ipc_perm* ipcp, int uid)
{
    if(uid/SEQ_MULTIPLIER != ipcp->seq)
        return 1;
    return 0;
}
```

可见，ipc_lock()中只用了标识号中的下标部分，而并没有检查其序号部分，所以要由 ipc_checkid()加以检查。此外，还要通过 ipcperms()检查当前进程是否有权向这个队列发送报文。注意在 ipc_lock()中调用了 spin_lock()将 ipc_ids 结构中的数组锁住，但是在正常条件下并没有解锁（只是在 out 为 0，即失败的情况下才调用了 spin_unlock()）。这个锁要在与 msg_lock()配对的 msg_unlock()，即 ipc_unlock()中才解开（ipc/util.h）：

==================== ipc/util.h 82 85 ====================

[sys_msgsnd()>msg_unlock()>ipc_unlock()]

```
extern inline void ipc_unlock(struct ipc_ids* ids, int id)
{
    spin_unlock(&ids->ary);
}
```

建议读者在往下读代码的过程中注意这个函数是在什么时候调用的以及为什么调用。

顺利通过了这些检查以后，就要检查报文队列的容量了。660 行的 if 语句中 msg->q_qbytes 为该报文队列的总容量，这个容量是在建立报文队列时设置好的，但是可以通过 MSGCTL 操作加以改变。如果当前报文的大小加上队列中当前总计的字节数 msg->q_qbytes 超出了这个容量，就暂时不能往队列里送了。此外，虽然总计字节数并未超过容量，但是队列中报文的个数 msg->q_qnum 却已经达到了 msg->q_qbytes（这说明队列中所有报文都只有一个字节，或有的报文长度为 0），就也不能往里送了。这时候，就要看系统调用参数中的 IPC_NOWAIT 是否为 1，是的话就出错返回，不然就要睡眠等待了。在进入睡眠之前，还要将一个 msg_sender 数据结构挂入报文队列中的 q_senders 链。这样，顺着每个报文队列的 q_senders 链就可以找到每个正在睡眠中等待着发送的进程。最后，还要在进入睡眠之前将报文队列解锁，使其它进程可以访问这个队列——要不然就没有进程可以从中读取报文。这里（668 行）的 ss_add()是个 inline 函数（ipc/msg.c）：

==================== ipc/msg.c 237 242 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_msgsnd()>ss_add()]

```
static inline void ss_add(struct msg_queue* msq, struct msg_sender* mss)
{
     mss->tsk=current;
     current->state=TASK_INTERRUPTIBLE;
     list_add_tail(&mss->list,&msq->q_senders);
}
```

而 msg_sender 结构的定义为（ipc/msg.c）：

==================== ipc/msg.c 45 49 ====================

```
/* one msg_sender for each sleeping sender */
struct msg_sender {
     struct list_head list;
     struct task_struct* tsk;
};
```

回过头去再看一下报文队列头 msg_queue 结构、报文头部 msg_msg 结构以及报文段头部 msg_msgseg 结构的定义，并将这些数据结构的定义综合起来，就可以形成这样一个图景（见图 6.8）。

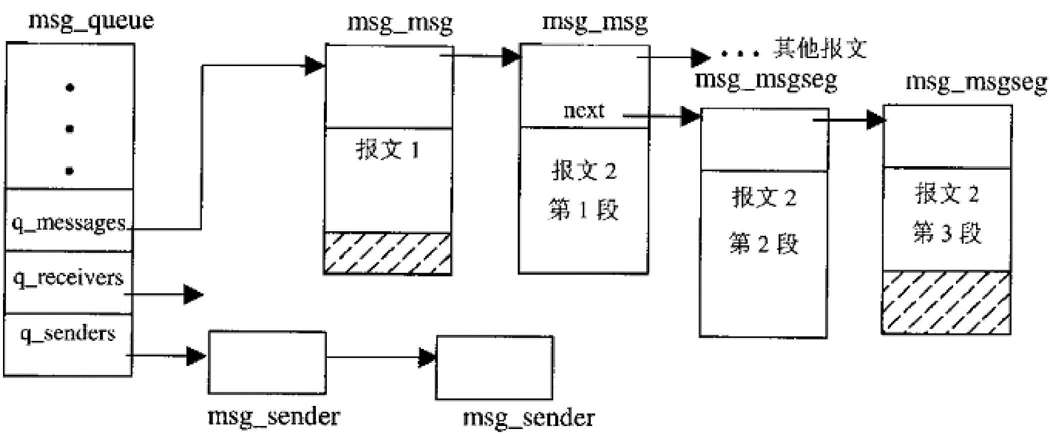


图 6.8　报文队列结构连接示意图

图中的第一个报文连同其头部 msg_msg 结构可以容纳在同一个页面中；而第二个报文则因为太大而分成三段，分处于三个页面中。队列的 q_senders 链中有两个进程正在睡眠中等待着向这个队列发送报文。队列中实际上还有一条 q_receivers 链，当有进程在睡眠中等待着从这个队列接收报文时，就将其 msg_receive 结构挂入这条链中。一般而言，当 q_senders 链中有进程在等着发送就说明队列中有报文，所以就不会有进程在 q_receivers 链中等待接收；反之亦然。可见，所谓报文队列实际上并不只是一个简单的队列，而是以报文的队列为主，包括了三个队列以及控制信息在内的一整套数据结构。

当有进程从这个队列中读取报文而腾出了一些空间时，正在等待着发送的进程就被唤醒，从前面 670 行的 schedule()返回。但是，腾出来的空间对于某个特定进程所欲发送的报文来说未必就已足够，而且正在等待发送的进程也可能不止一个，甚至，在发送进程睡眠的期间，其它条件也可能已经发生了变化。所以，这时候还要再来一轮对各种条件的检查，而在此期间又得将该报文队列锁住。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在系统调用中，一个进程从睡眠中醒来时通常都要检查是否有信号在等待处理，如有的话就提前结束本次系统调用的执行。也就是说，在这种情况下把对信号的处理放在更为优先的地位。同时，由于系统调用“流产”了，所以返回出错代码-EINTR，表示系统调用的执行被中途打断了。我们以前讲过，内核在完成对信号的处理以后要检查系统调用返回的出错代码，如果遇上-EINTR 就会安排重新执行一次流产的系统调用。

如果没有信号在等待处理，则通过 683 行的 goto 语句转回标号 retry 处重新检查一遍。

就这样，在正常条件下，最终队列中总会有足够的空间让等待中的进程发送其报文，那时候程序就往下走了（ipc/msg.c）：

==================== ipc/msg.c 686 706 ====================

[sys_msgsnd()]

```
	if(!pipelined_send(msq,msg)) {
		/* noone is waiting for this message, enqueue it */
		list_add_tail(&msg->m_list,&msq->q_messages);
		msq->q_cbytes += msgsz;
		msq->q_qnum++;
		atomic_add(msgsz,&msg_bytes);
		atomic_inc(&msg_hdrs);
	}

	err = 0;
	msg = NULL;
	msq->q_lspid = current->pid;
	msq->q_stime = CURRENT_TIME;

out_unlock_free:
	msg_unlock(msqid);
out_free:
	if(msg!=NULL)
		free_msg(msg);
	return err;
}
```

从概念上讲，此时可以把待发送的报文挂入报文队列，然后唤醒（可能）正在等待从此队列接收报文的进程就完事了。可是，实际上这个过程是可以优化的，凶为如果有进程正在等待接收，就不必将报文链入队列再由接收进程随后再来将报文从队列中脱链了。也就是说，只有当没有进程在等待接收时才需要将报文挂入队列。代码中的 pipelined_send()正是为此而设计的（ipc/msg.c）。

==================== ipc/msg.c 600 624 ====================

[sys_msgsnd()>pipelined_send()]

```
int inline pipelined_send(struct msg_queue* msq, struct msg_msg* msg)
{
	struct list_head* tmp;

	tmp = msq->q_receivers.next;
	while (tmp != &msq->q_receivers) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
606             struct msg_receiver* msr;
607             msr = list_entry(tmp,struct msg_receiver,r_list);
608             tmp = tmp->next;
609             if(testmsg(msg,msr->r_msgtype,msr->r_mode)) {
610                  list_del(&msr->r_list);
611                  if(msr->r_maxsize < msg->m_ts) {
612                       msr->r_msg = ERR_PTR(-E2BIG);
613                       wake_up_process(msr->r_tsk);
614                  } else {
615                       msr->r_msg = msg;
616                       msq->q_lspid = msr->r_tsk->pid;
617                       msq->q_rtime = CURRENT_TIME;
618                       wake_up_process(msr->r_tsk);
619                       return 1;
620                  }
621             }
622        }
623        return 0;
624   }
```

报文队列的 q_receivers 链中聚集着正在睡眠等待从本队列接收报文的进程（如果有的话）。数据结构 msg_receiver 与 msg_sender 有所不同（ipc/msg.c）：

==================== ipc/msg.c 33 43 ====================

```
/* one msg_receiver structure for each sleeping receiver */
struct msg_receiver {
     struct list_head r_list;
     struct task_struct* r_tsk;

     int r_mode;
     long r_msgtype;
     long r_maxsize;

     struct msg_msg* volatile r_msg;
};
```

其中的指针 r_msg 就是在上述情况下用于报文交接的。函数 pipelined_send()在 q_receivers 链中从头开始，逐个检查等待中的进程里想要接收的报文种类与模式是否与到来的报文相符。若相符，则进一步检查其缓冲区是否够用，不够用就将该进程唤醒令其出错返回。此过程直至碰到第一个所有条件全部相符的接收进程，然后将到来的报文交给这个进程（见 615 行），并将其唤醒。

如果遍历了 q_receivers 链而没有发现任何一个等待中的进程满足条件，则 pipelined_send()返回 0，那样，在 sys_msgsnd()中就要将报文挂入队列中了（见 688 行）。

6.6.3 库函数 msgrcv()——报文接收

再来看报文的接收，我们还是逐段地往下看（ipc/msg.c）：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== ipc/msg.c 727 743 ====================

```
727 asmlinkage long sys_msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz,
728                   long msgtyp, int msgflg)
729 {
730      struct msg_queue *msq;
731      struct msg_receiver msr_d;
732      struct list_head* tmp;
733      struct msg_msg* msg, *found_msg;
734      int err;
735      int mode;
736
737      if (msqid < 0 || (long) msgsz < 0)
738           return -EINVAL;
739      mode = convert_mode(&msgtyp,msgflg);
740
741      msq = msg_lock(msqid);
742      if(msq==NULL)
743           return -EINVAL;
```

这一段与 sys_msgsnd()的开头相似，739 行函数 convert_mode()根据参数 msgtyp 和 msgflg，归纳出接收报文时所遵循的准则（ipc/msg.c）：

==================== ipc/msg.c 708 725 ====================

[sys_msgrcv()>convert_mode()]

```
int inline convert_mode(long* msgtyp, int msgflg)
{
     /*
      *  find message of correct type.
      *  msgtyp = 0 => get first.
      *  msgtyp > 0 => get first message of matching type.
      *  msgtyp < 0 => get message with least type must be < abs(msgtype).
      */
     if(*msgtyp==0)
          return SEARCH_ANY;
     if(*msgtyp<0) {
          *msgtyp=-(*msgtyp);
          return SEARCH_LESSEQUAL;
     }
     if(msgflg & MSG_EXCEPT)
          return SEARCH_NOTEQUAL;
     return SEARCH_EQUAL;
}
```

程序中的注解已经讲得很清楚，程序也很简单。

函数 msg_lock()的代码已经在前面看到过了，它根据报文队列标识号找到具体的队列并将其上锁。我们继续往下看。读者可以看到，sys_msgrcv()在程序结构上与 sys_msgsnd()是相似的（ipc/msg.c）：

==================== ipc/msg.c 744 764 ====================

[sys_msgrcv()]

```
744 retry:
745     err=-EACCES;
746     if (ipcperms (&msq->q_perm, S_IRUGO))
747         goto out_unlock;
748 
749     tmp = msq->q_messages.next;
750     found_msg=NULL;
751     while (tmp != &msq->q_messages) {
752         msg = list_entry(tmp,struct msg_msg,m_list);
753         if(testmsg(msg,msgtyp,mode)) {
754             found_msg = msg;
755             if(mode == SEARCH_LESSEQUAL && msg->m_type != 1) {
756                 found_msg=msg;
757                 msgtyp=msg->m_type-1;
758             } else {
759                 found_msg=msg;
760                 break;
761             }
762         }
763         tmp = tmp->next;
764     }
```

首先是检查当前进程对队列的访问权限（746 行）。函数 ipcperms()的代码已经在以前看到过了，所不同的是这里的参数为 S_IRUGO。因为从队列接收相当于从文件读；而在 sys_msgsnd()中使用的参数则为 S_IWUGO，因为那相当于向文件写。

然后就是逐个检查已经在队列中的报文了。注意 752 行的宏操作 list_entry()并不是将报文从队列中脱链，而只是根据队列中的当前项找到指向其所在数据结构的指针。我们在第 1 章中曾介绍过通用的队列操作，读者如果忘了可以回过头去看一下。接着，753 行中的函数 testmsg()的代码如下（ipc/msg.c）：

==================== ipc/msg.c 578 598 ====================

[sys_msgrcv()>testmsg()]

```
static int testmsg(struct msg_msg* msg,long type,int mode)
{
    switch(mode)
    {
        case SEARCH_ANY:
            return 1;
        case SEARCH_LESSEQUAL:
            if(msg->m_type <=type)
                return 1;
            break;
        case SEARCH_EQUAL:
            if(msg->m_type == type)
                return 1;
            break;
        case SEARCH_NOTEQUAL:
            if(msg->m_type != type)
                return 1;
```

```
                break;
        }
        return 0;
}
```

当接收的准则（mode）为 SEARCH_LESSEQUAL 时，testmsg()在这里所检查的条件是 msg->m_type <= msgtyp，也就是该报文的类型值小于或等于作为参数传过来的 msgtyp。然而，SEARCH_LESSEQUAL 所要求的实际上是已经到达的报文中类型值为最小者，而类型值最小是 1。所以，虽然找到了一个满足 msg->m_type <= msgtyp 这个条件的报文，但却未必是最佳选择。因此，只是将此报文作为迄今为止的最佳候选，而将 msgtyp 的值减小到比这个报文的类型值更小（见 757 行），看看还能不能找到更小的。不然的话，当接收的准则为 SEARCH_ANY 或其它时，则总是找队列中第一个满足条件的报文，而且一旦发现了一个就不用再往下找了（见 760 行的 break 语句）。这样，当 751 行开始的 while 循环结束时，只要有一个报文符合接收准则和类型，found_msg 就指向这个报文。再往下看（ipc/msg.c）：

==================== ipc/msg.c 765 788 ====================

[sys_msgrcv()]

```
765         if(found_msg) {
766             msg=found_msg;
767             if ((msgsz < msg->m_ts) && !(msgflg & MSG_NOERROR)) {
768                 err=-E2BIG;
769                 goto out_unlock;
770             }
771             list_del(&msg->m_list);
772             msq->q_qnum--;
773             msq->q_rtime = CURRENT_TIME;
774             msq->q_lrpid = current->pid;
775             msq->q_cbytes -= msg->m_ts;
776             atomic_sub(msg->m_ts,&msg_bytes);
777             atomic_dec(&msg_hdrs);
778             ss_wakeup(&msq->q_senders,0);
779             msg_unlock(msqid);
780 out_success:
781             msgsz = (msgsz > msg->m_ts) ? msg->m_ts : msgsz;
782             if (put_user (msg->m_type, &msgp->mtype) ||
783                store_msg(msgp->mtext, msg, msgsz)) {
784                    msgsz = -EFAULT;
785             }
786             free_msg(msg);
787             return msgsz;
788         } else
```

从队列中找到了符合接收准则和类型的报文，还不一定就能够接收这个报文，还得要看用户程序所提供的缓冲区空间是否足够大。如果缓冲区空间 msgsz 不够大，而用户又不允许在报文的尾部截掉一块，那就只好出错返回了。反之，就可以接收这个报文，也就是将其从队列中脱链（见 771 行），并相应调整和设置一些用于控制和统计的变量。前面讲过，当队列中已经有报文存在时，如果有进程要向此队列发送报文，就有可能因队列的容量不足而只好在队列的发送进程链中等待。现在既然从队列

中接收了一个报文，就腾出了一些空间，所以要调用 ss_wakeup()，顺着这个链将正在睡眠中的进程全部唤醒（ipc/msg.c）：

==================== ipc/msg.c 250 264 ====================

[sys_msgrcv()>ss_wakeup()]

```
250 static void ss_wakeup(struct list_head* h, int kill)
251 {
252      struct list_head *tmp;
253 
254      tmp = h->next;
255      while (tmp != h) {
256           struct msg_sender* mss;
257 
258           mss = list_entry(tmp,struct msg_sender,list);
259           tmp = tmp->next;
260           if(kill)
261                mss->list.next=NULL;
262           wake_up_process(mss->tsk);
263      }
264 }
```

注意这里在将每个 msg_sender 结构脱链以后并不释放其空间，因为这个数据结构是作为局部量分配在相应进程的系统空间堆栈上的。读者不妨回过去看一下 sys_msgsnd()的代码。

最后，要将实际接收的报文类型通过 put_user()送回用户空间，并通过 store_msg()将接收到的报文按实际接收的长度（见 781 行）复制到用户空间，然后将系统空间中的报文（缓冲区）释放。

那么，如果此时报文队列中尚无报文可供接收呢？再往下看（ipc/msg.c）：

==================== ipc/msg.c 788 844 ====================

[sys_msgrcv()]

```
788      } else
789      {
790           struct msg_queue *t;
791           /* no message waiting. Prepare for pipelined
792            * receive.
793            */
794           if (msgflg & IPC_NOWAIT) {
795                err=-ENOMSG;
796                goto out_unlock;
797           }
798           list_add_tail(&msr_d.r_list,&msq->q_receivers);
799           msr_d.r_tsk = current;
800           msr_d.r_msgtype = msgtyp;
801           msr_d.r_mode = mode;
802           if(msgflg & MSG_NOERROR)
803                msr_d.r_maxsize = INT_MAX;
804            else
805                msr_d.r_maxsize = msgsz;
806           msr_d.r_msg = ERR_PTR(-EAGAIN);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        current->state = TASK_INTERRUPTIBLE;
        msg_unlock(msqid);

        schedule();
        current->state = TASK_RUNNING;

        msg = (struct msg_msg*) msr_d.r_msg;
        if(!IS_ERR(msg))
            goto out_success;

        t = msg_lock(msqid);
        if(t==NULL)
            msqid=-1;
        msg = (struct msg_msg*)msr_d.r_msg;
        if(!IS_ERR(msg)) {
            /* our message arived while we waited for
             * the spinlock. Process it.
             */
            if(msqid!=-1)
                msg_unlock(msqid);
            goto out_success;
        }
        err = PTR_ERR(msg);
        if(err == -EAGAIN) {
            if(msqid==-1)
                BUG();
            list_del(&msr_d.r_list);
            if (signal_pending(current))
                err=-EINTR;
             else
                goto retry;
        }
    }
out_unlock:
    if(msqid!=-1)
        msg_unlock(msqid);
    return err;
}
```

如果调用参数中的 IPC_NOWAIT 标志为 1，那就立即返回了，出错代码为-ENOMSG。否则，就要睡眠等待了。当然，入睡前要将报文队列解锁。

就像在睡眠时等待发送时一样，在睡眠等待接收时也要将一个代表当前进程的 msg_receiver 数据结构链入报文队列中的 q_receiver 队列。我们已经在前面看到过这种数据结构的定义。与 msg_sender 不同的是，在 msg_receiver 结构中还记录着所欲接收的报文类型、缓冲区大小以及接收的准则。此外，这个结构中还有一个用来交接报文的指针 r_msg。所有这些结构成分的设置都是为了一旦有进程要向此报文队列发送报文时可以抄近路（见 pipelined_send()的代码）。

一般来说，当前进程一旦睡下，就要等到有进程通过 pipelined_send()向这个队列发送报文，并且

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

选择这个进程作为接收进程时才会被唤醒，因此要到那时候才能从 810 行的 schedule()返回。结合 pipelined_send()的代码，可以看到当前进程在被唤醒的时候有两种可能的情况。一种是已经接收到了符合要求的报文，此时 msr_d.r_msg 为指向该报文的指针。另一种是报文的类型相符，但是接收进程的缓冲区太小而不能接收，此时 msr_d.r_msg 持有出错的代码-E2BIG（见 pipelined_send()中的 612 行）。然而，在另外一种情况下也会唤醒睡眠中的进程，那就是如果这个进程接收到了一个信号。由于 msr_d.r_msg 在进入睡眠前，已被预设为-EAGAIN（见 806 行），所以在这种情况下 msr_d.r_msg 中也是一个出错代码。因此，当进程从 schedule()返回时，若 msr_d.r_msg 中不是出错代码就表示接收成功了（见 827 行）。

如果是出错代码呢？这里要仔细往下看了。原作者在这里加了注释，但可能不是很容易看懂。首先是对报文队列加锁。对于这一点应该是好理解的，只要看到 837 行的 goto 语句就能明白，当前进程还要回到 744 行的标号 retry 处开始新一轮的接收操作，所以要将报文队列锁住。但是，msg_lock()中调用的 spin_lock()可能隐藏着等待，因为有可能另一个进程（在另一个 CPU 上运行）已经抢先一步把队列锁住了。另一方面，还要看到，如果当前进程是因为接收到信号而被唤醒，则其 msg_receiver 结构 msr_d 仍留在报文队列的 q_receivers 链中。这样，在进程被唤醒以后，直到在 817 行的 msg_lock()中成功地将报文队列锁住之前，仍有可能接收到其它进程通过 pipelined_send()发来的报文。正因为这样，在 msg_lock()之后还要再检查一下 msr_d.r_msg。如果它变成了一个报文指针，那么也是接收成功了。虽然此时有信号在等待着处理，但由于本次系统调用的主体已经完成，所以还是转向 out_success 先将接收到的报文复制到用户空间中。反正那以后很快就要从系统调用返回，到那时候再来处理信号也不迟。但是，如果 msr_d.r_msg 仍是出错代码，那就又要分析了。在 806 行，已经将 msr_d.r_msg 预设成-EAGAIN，只有 pipelined_send()，才会在唤醒一个正在等待接收的进程前改变其 msr_d.r_msg 的内容。所以，如果出错代码为-E2BIG，就说明当前进程肯定是由某个发送进程在 pipelined_send()中唤醒的。只要出错代码是-E2BIG（只要不是-EAGAIN 就必然是-E2BIG），就出错返回了。当然，在返回前要将报文队列解锁（见 842 行）。至于有可能正在等待处理的信号，则在从系统调用返回之前还会有检查并处理。反之，如果出错代码为-EAGAIN，则说明当前进程不是由 pipelined_send()唤醒的，所以要将本进程的 msg_receiver 结构脱链。然后再看到底是否有信号在等着要处理。如果有信号等待处理，就将返回值设成-EINTR 并且提前返回；否则，如果没有信号在等待处理的话，就跳转回 retry 开始新一轮的报文接收。注意，由于 msg_receiver 结构 msr_d 是作为局部量分配在堆栈上的，所以不用（而且不可）释放其空间。

6.6.4 库函数 msgctl()——报文机制的控制与设置

前面讲过，命名管道和无名管道的缺点之一是缺乏对管道的控制手段，也缺乏获取其状态信息（例如，有多少个字节已经在管道中等待着读取）的手段。MSGCTL 操作正是为报文队列提供了这样的手段。内核中的函数 sys_msgctl()的界面为（include/linux/msg.h）：

==================== include/linux/msg.h 69 69 ====================

```
69   asmlinkage long sys_msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

其中参数 cmd 为具体的命令码，定义于 include/linux/ipc.h 中：

==================== include/linux/ipc.h 34 41 ====================

```
/*
 * Control commands used with semctl, msgctl and shmctl
 * see also specific commands in sem.h, msg.h and shm.h
 */
#define IPC_RMID 0     /* remove resource */
#define IPC_SET  1     /* set ipc_perm options */
#define IPC_STAT 2     /* get ipc_perm options */
#define IPC_INFO 3     /* see ipcs */
```

这些命令码并不是专为报文队列而设置的，也适用于 SysV IPC 的其它两种机制。对于具体的机制则可能还要再补充若干专用的命令。就报文队列而言，在 include/linux/msg.h 中定义了两个专用命令码：

==================== include/linux/msg.h 6 8 ====================

```
/* ipcs ctl commands */
#define MSG_STAT 11
#define MSG_INFO 12
```

在为 SysV IPC 设置的这些命令中，IPC_RMID 用来撤消一个标识号，对报文队列而言也就是撤消一个报文队列，其作用相当于文件系统中的“关闭文件”。IPC_SET 用来改变相应 IPC 设施的各种状态和属性。而 IPC_STAT 和 IPC_INFO 则分别用来获取关于相关设施的状态或统计信息。

调用参数 buf 为一个 msqid_ds 结构指针。这个结构使用于 IPC_STAT 和 IPC_SET，是在 include/linux/msg.h 中定义的：

==================== include/linux/msg.h 14 29 ====================

```
/* Obsolete, used only for backwards compatibility and libc5 compiles */
struct msqid_ds {
     struct ipc_perm msg_perm;
     struct msg *msg_first;      /* first message on queue,unused  */
     struct msg *msg_last;       /* last message in queue,unused */
     __kernel_time_t msg_stime;  /* last msgsnd time */
     __kernel_time_t msg_rtime;  /* last msgrcv time */
     __kernel_time_t msg_ctime;  /* last change time */
     unsigned long  msg_lcbytes; /* Reuse junk fields for 32 bit */
     unsigned long  msg_lqbytes; /* ditto */
     unsigned short msg_cbytes;  /* current number of bytes on queue */
     unsigned short msg_qnum;    /* number of messages in queue */
     unsigned short msg_qbytes;  /* max number of bytes on queue */
     __kernel_ipc_pid_t msg_lspid;    /* pid of last msgsnd */
     __kernel_ipc_pid_t msg_lrpid;    /* last receive pid */
};
```

代码作者在注释中说这个数据结构已经过时，只是为了兼容才保留着。新的数据结构是 msqid64_ds，定义于 include/asm-i386/msgbuf.h 中，显然是为 64 位系统结构准备的。

==================== include/asm-i386/msgbuf.h 4 29 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 * The msqid64_ds structure for i386 architecture.
 * Note extra padding because this structure is passed back and forth
 * between kernel and user space.
 *
 * Pad space is left for:
 * - 64-bit time_t to solve y2038 problem
 * - 2 miscellaneous 32-bit values
 */

struct msqid64_ds {
	struct ipc64_perm msg_perm;
	__kernel_time_t msg_stime;	/* last msgsnd time */
	unsigned long __unused1;
	__kernel_time_t msg_rtime;	/* last msgrcv time */
	unsigned long __unused2;
	__kernel_time_t msg_ctime;	/* last change time */
	unsigned long __unused3;
	unsigned long  msg_cbytes;	/* current number of bytes on queue */
	unsigned long  msg_qnum;	/* number of messages in queue */
	unsigned long  msg_qbytes;	/* max number of bytes on queue */
	__kernel_pid_t msg_lspid;	/* pid of last msgsnd */
	__kernel_pid_t msg_lrpid;	/* last receive pid */
	unsigned long  __unused4;
	unsigned long  __unused5;
};
```

不过，我们从前面 sys_msgctl()的调用参数表中可以看到在系统调用界面上还在用着 msqid_ds。结构中 msg_cbytes 和 msg_lcbytes，以及 msg_qbytes 和 msg_lqbytes 在逻辑上是相同的，只不过一为无符号短整数，一为无符号长整数。

当命令码为 IPC_INFO 时，则 buf 指向一个 msginfo 结构（include/linux/msg.h）：

==================== include/linux/msg.h 40 50 ====================

```
/* buffer for msgctl calls IPC_INFO, MSG_INFO */
struct msginfo {
	int msgpool;
	int msgmap;
	int msgmax;
	int msgmnb;
	int msgmni;
	int msgssz;
	int msgtql;
	unsigned short  msgseg;
};
```

建议读者结合 sys_msgsnd()和 sys_msgrcv()的代码以及 msg_msg 结构的定义，看看这两个数据结构中诸成分的用途。

函数 sys_msgctl()的代码不短，但是很简单。所以我们把它留给读者自己阅读，也不在这里列出其

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

代码了。但这里只看其中的一个片段（ipc/msg.c）：

==================== ipc/msg.c 539 566 ====================

[sys_msgctl()]

```
	switch (cmd) {
	case IPC_SET:
	{
		if (setbuf.qbytes > msg_ctlmnb && !capable(CAP_SYS_RESOURCE))
			goto out_unlock_up;
		msq->q_qbytes = setbuf.qbytes;

		ipcp->uid = setbuf.uid;
		ipcp->gid = setbuf.gid;
		ipcp->mode = (ipcp->mode & ~S_IRWXUGO) |
			(S_IRWXUGO & setbuf.mode);
		msq->q_ctime = CURRENT_TIME;
		/* sleeping receivers might be excluded by
		 * stricter permissions.
		 */
		expunge_all(msq,-EAGAIN);
		/* sleeping senders might be able to send
		 * due to a larger queue size.
		 */
		ss_wakeup(&msq->q_senders,0);
		msg_unlock(msqid);
		break;
	}
	case IPC_RMID:
		freeque (msqid);
		break;
	}
	err = 0;
```

这里的 setbuf 是从用户空间复制过来的 msqid_ds 数据结构。每当通过 IPC_SET 改变一个报文队列的有关参数时，由于参数的改变而需要做两件事。一件事是使正在等待此队列接收报文的进程（如果有的话）都出错返回，出错代码为-EAGAIN。这是通过 expunge_all()完成的。第二件事是将正在等待向此队列发送报文的进程（如果有的话）全部唤醒，让它们开始新一轮尝试。这是通过 ss_wakeup()完成的，其代码前面已经看到过。此处给出函数 expunge_all()的代码（ipc/msg.c）：

==================== ipc/msg.c 266 279 ====================

[sys_msgctl()>expunge_all()]

```
static void expunge_all(struct msg_queue* msq, int res)
{
	struct list_head *tmp;

	tmp = msq->q_receivers.next;
	while (tmp != &msq->q_receivers) {
		struct msg_receiver* msr;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

            msr = list_entry(tmp,struct msg_receiver,r_list);
            tmp = tmp->next;
            msr->r_msg = ERR_PTR(res);
            wake_up_process(msr->r_tsk);
        }
}
```

命令 RMID 则是通过 freeqeue()完成的，现在其代码对于读者应该已经很简单了（ipc/msg.c）：

==================== ipc/msg.c 281 301 ====================

[sys_msgctl()>freeque()]

```
static void freeque (int id)
{
        struct msg_queue *msq;
        struct list_head *tmp;

        msq = msg_rmid(id);

        expunge_all(msq,-EIDRM);
        ss_wakeup(&msq->q_senders,1);
        msg_unlock(id);

        tmp = msq->q_messages.next;
        while(tmp != &msq->q_messages) {
            struct msg_msg* msg = list_entry(tmp,struct msg_msg,m_list);
            tmp = tmp->next;
            atomic_dec(&msg_hdrs);
            free_msg(msg);
        }
        atomic_sub(msq->q_cbytes, &msg_bytes);
        kfree(msq);
}
```

同样，freeque()将所有正在等待接收报文的进程都从接收队列里脱链并唤醒，让它们出错返回。并通过 ss_wakeup()，将所有正在等待发送报文的进程都从发送队列里脱链并唤醒，也让它们出错返回。然后将队列中所有报文都脱链并释放其空间。最后连报文队列头也予以释放。

代码中的 msg_rmid()是个宏定义（ipc/msg.c）：

==================== ipc/msg.c 96 96 ====================

```
#define msg_rmid(id)    ((struct msg_queue*)ipc_rmid(&msg_ids,id))
```

函数 ipc_rmid()的代码在 ipc/util.c 中，也很简单：

==================== ipc/util.c 173 205 ====================

[sys_msgctl()>freeque()>msg_rmid()>ipc_rmid()]

```
/**
 *  ipc_rmid -    remove an IPC identifier
 *  @ids: identifier set
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   @id: Identifier to remove
 *
 *   The identifier must be valid, and in use. The kernel will panic if
 *   fed an invalid identifier. The entry is removed and internal
 *   variables recomputed. The object associated with the identifier
 *   is returned.
 */

struct kern_ipc_perm* ipc_rmid(struct ipc_ids* ids, int id)
{
     struct kern_ipc_perm* p;
     int lid = id % SEQ_MULTIPLIER;
     if(lid > ids->size)
          BUG();
     p = ids->entries[lid].p;
     ids->entries[lid].p = NULL;
     if(p==NULL)
          BUG();
     ids->in_use--;

     if (lid == ids->max_id) {
          do {
               lid--;
               if(lid == -1)
                    break;
          } while (ids->entries[lid].p == NULL);
          ids->max_id = lid;
     }
     return p;
}
```

这里有个问题，被唤醒的进程会做些什么呢？读者不妨回过头去看看 sys_msgsnd()和 sys_msgrcv()中的代码（msg.c 中的 673 行和 817 行），睡眠中的进程被唤醒，并且被调度运行而从 schedule()返回后，还要再调用 msg_lock()，但是那时候就会返回 NULL，从而从 sys_msgsnd()或 sys_msgrcv()失败返回了。

6.7 共享内存

共享内存，顾名思义就是两个或更多个进程可以访问同一块内存区间，使得一个进程对这块空间中某个单元内容的改变可以为其它进程所“看”到。共享内存是针对（命名或无名）管道以及其它机制运行效率比较低的缺陷而设计的。虽然报文队列比之管道有了很大的改进，但是从运行效率的角度来说却并无什么明显的不同。而共享内存，则由于参加共享的各个进程就像访问普通内存一样地访问所共享的内存区间，其运行时的效率可以很高。对于某些运行效率显得很关键的应用来说，可能会觉得管道或报文队列的速度太慢，所以宁愿放弃一些由这些机制所提供的好处，而采用共享内存作为进程间通信的手段。不过，应该指出，共享内存是一种很低级（与物理层很贴近）的通信机制，所提供

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的功能是很有限的，所以使用时要特别小心。

一般来说，一种进程间通信机制常常附加地提供一些进程间同步和互斥的功能，传递的内容也得到一定程度的缓冲。以管道为例，从管道读的进程在管道中无内容可读时就会进入睡眠等待。同样地，向管道写的进程在管道被写满时也会睡眠等待。虽然这两个过程从宏观上说是并发的，但是从微观上，也就是从系统调用的内部实现层次上说，却是独占的、互斥的，因而是“原子的”。所以读和写的过程通过内核加以“串行化”而不会互相干扰，同时，写入管道的内容也得到了缓冲。可是共享内存就不同了。不同进程读 / 写一块内存空间的操作本身就是微观的、直接的，并不通过系统调用来进行。这样，就失去了由内核保证互斥性的可能，进程间也不会因此而自动地得到同步，并且所写的内容在全部完成前就立即可以部分地为其它进程所“看”到。举例来说，如果进程 A 向一块共享内存空间写一字符串，在写了一半时就因中断而引起调度，于是当进程 B 在进程 A 尚未完成整个字符串的写入前就来读，那就会读出这样一个字符串：其前半部是进程 A 新写入的，可是后半部却是以前某个时候由其它进程写入的。所以，共享内存通常要与 Sys V IPC 中的另一个机制“信号量”结合使用，这样才能达到进程间的同步与互斥。

在内核中，共享内存机制的四种操作 SHMGET、SHMAT、SHMDT 和 SHMCTL，即应用程序设计界面上的库函数 shmget()、shmat()、shmdt()和 shmctl()，分别是由 sys_shmget()、sys_shmat()、sys_shmdt()和 sys_shmctl()实现的。与报文队列相似，参加共享内存的进程之一首先要创建一个共享内存区，然后其它进程通过一个共同的键值取得它的标识号。得到了一个共享内存区的标识号以后，每个进程就可以将此共享内存区“挂靠”（attach）到（实际上是映射到）它的虚存空间，然后就可以像访问一般内存一样地访问这块共享内存区了。要退出对此共享区间的共享时，则可以将其“脱钩”（dettach），也就是解除映射。有关的代码基本上都在文件 ipc/shm.c 中。

6.7.1 库函数 shmget()——共享内存区的创建与寻找

库函数 shmget()，即 ipc()系统调用的 SHMGET 操作是由 sys_shmget()完成的（ipc/shm.c）：

==================== ipc/shm.c 224 253 ====================

```
asmlinkage long sys_shmget (key_t key, size_t size, int shmflg)
{
	struct shmid_kernel *shp;
	int err, id = 0;

	down(&shm_ids.sem);
	if (key == IPC_PRIVATE) {
		err = newseg(key, shmflg, size);
	} else if ((id = ipc_findkey(&shm_ids, key)) == -1) {
		if (!(shmflg & IPC_CREAT))
			err = -ENOENT;
		else
			err = newseg(key, shmflg, size);
	} else if ((shmflg & IPC_CREAT) && (shmflg & IPC_EXCL)) {
		err = -EEXIST;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    } else {
        shp = shm_lock(id);
        if(shp==NULL)
            BUG();
        if (shp->shm_segsz < size)
            err = -EINVAL;
        else if (ipcperms(&shp->shm_perm, shmflg))
            err = -EACCES;
        else
            err = shm_buildid(id, shp->shm_perm.seq);
        shm_unlock(id);
    }
    up(&shm_ids.sem);
    return err;
}
```

与前一节中的 sys_msgget()作一比较，就可发现二者基本相同。所不同的是这里所调用的是 newseg()，因为是创建新的共享内存区，而不是报文队列。像报文队列一样，内核中也有个全局的 ipc_ids 数据结构 shm_ids：

==================== ipc/shm.c 48 48 ====================

```
static struct ipc_ids shm_ids;
```

如前所述，ipc_ids 数据结构中有个指针 entries，指向一个 ipc_id 结构数组，而每个 ipc_id 结构中则有个指针 p，指向一个 kern_ipc_perm 数据结构。所不同的是，在报文队列机制中，kern_ipc_perm 数据结构的“宿主”是 msg_queue 数据结构，代表着一个报文队列；而在共享内存机制中则是 shmid_kernel 数据结构，代表着一个共享内存区。等一下读者就会看到它的定义。

同样键值 IPC_PRIVATE，即 0，是特殊的，它表示要分配一个共享内存区供本进程专用。其它键值则表示要创建或寻找的是“共享”内存区。而标志位 IPC_CREAT 则表示目的在于创建。

函数 newseg()创建一块共享内存区（ipc/shm.c）：

==================== ipc/shm.c 173 222 ====================

[sys_shmget()>newseg()]

```
static int newseg (key_t key, int shmflg, size_t size)
{
    int error;
    struct shmid_kernel *shp;
    int numpages = (size + PAGE_SIZE -1) >> PAGE_SHIFT;
    struct file * file;
    char name[13];
    int id;

    if (size < SHMMIN || size > shm_ctlmax)
        return -EINVAL;

    if (shm_tot + numpages >= shm_ctlall)
        return -ENOSPC;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    shp = (struct shmid_kernel *) kmalloc (sizeof (*shp), GFP_USER);
    if (!shp)
        return -ENOMEM;
    sprintf (name, "SYSV%08x", key);
    file = shmem_file_setup(name, size);
    error = PTR_ERR(file);
    if (IS_ERR(file))
        goto no_file;

    error = -ENOSPC;
    id = shm_addid(shp);
    if(id == -1)
        goto no_id;
    shp->shm_perm.key = key;
    shp->shm_flags = (shmflg & S_IRWXUGO);
    shp->shm_cprid = current->pid;
    shp->shm_lprid = 0;
    shp->shm_atim = shp->shm_dtim = 0;
    shp->shm_ctim = CURRENT_TIME;
    shp->shm_segsz = size;
    shp->shm_nattch = 0;
    shp->id = shm_buildid(id,shp->shm_perm.seq);
    shp->shm_file = file;
    file->f_dentry->d_inode->i_ino = shp->id;
    file->f_op = &shm_file_operations;
    shm_tot += numpages;
    shm_unlock (id);
    return shp->id;

no_id:
    fput(file);
no_file:
    kfree(shp);
    return error;
}
```

首先根据共享区的大小计算出所需的存储页面数量 numpages，接着是对资源数量的一些检查。代码中的 shm_tot 和 shm_ctlall 都是全局量，分别用来记录当前已经用于共享内存机制的页面数及其上限。另一个全局量 shm_ctlmax 给出了对每个共享内存区大小的限制。每个共享内存区都有个区名，由前缀 SYSV 和键值的 8 位 16 进制数表示构成。

在内核中，每个共享内存区都由一个控制结构，即 shmid_kernel 数据结构代表，定义于 ipc/shm.c：

==================== ipc/shm.c 29 41 ====================

```
struct shmid_kernel /* private to the kernel */
{
    struct kern_ipc_perm    shm_perm;
    struct file *      shm_file;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        int             id;
        unsigned long       shm_nattch;
        unsigned long       shm_segsz;
        time_t              shm_atim;
        time_t              shm_dtim;
        time_t              shm_ctim;
        pid_t               shm_cprid;
        pid_t               shm_lprid;
};
```

显然，这个数据结构与用于报文队列的 msq_queue 相似，它的第一个结构成分也是个 kern_ipc_perm 结构。可是，这里有个 file 结构指针 shm_file。为什么在代表共享内存区的数据结构中有指向 file 结构的指针呢？

共享内存区中的页面也和普通的页面一样，受到内存页面管理机制的调度，根据实际的需要而换出／换入。不同的是，普通的页面都换出到通用的页面交换设备（或文件）上；而共享内存区则各自设立专用的映射文件，以共享内存区的区名作为文件名。这样，就把共享内存区与第 2 章中的文件映射联系在一起了。文件映射机制成了实现共享内存区的基础，而共享内存区则成了文件映射的一项应用。因此，对于每个进程，参与共享的内存区就好像已经建立的文件映射一样，可以通过特殊文件系统/proc 中的路径/proc/<pid>/maps 观察其状态（这里<pid>表示具体的进程号）。

为此，内核中专门设置了一种特殊文件系统“ shm”，其类型为 shmem_fs_type，定义于 mm/shmem.c：

==================== mm/shmem.c 702 702 ====================

```
static DECLARE_FSTYPE(shmem_fs_type, "shm", shmem_read_super, FS_LITTER);
```

编译时会把宏定义 DECLARE_FSTYPE 展开如下（参阅“文件系统”一章中的有关内容）：

```
struct  file_system_type shmem_fs_type = {
name:    “shm”,
reda_super: shmem_read_super,
fs_flags:  FS_LITTER,
owner:     THIS_MODULE,
kern_mnt:  NULL
）
```

系统在初始化时会通过 kern_mount()安装这个特殊文件系统，并在 devfs 特殊文件系统中创建起一个子目录 shm。这是在 init_shmem_fs()中完成的，代码见 mm/shmem.c：

==================== mm/shmem.c 704 723 ====================

```
static int __init init_shmem_fs(void)
{
        int error;
        struct vfsmount * res;

        if ((error = register_filesystem(&shmem_fs_type))) {
             printk (KERN_ERR "Could not register shmem fs\n");
             return error;
        }
```

```
713
714         res = kern_mount(&shmem_fs_type);
715         if (IS_ERR (res)) {
716              printk (KERN_ERR "could not kern_mount shmem fs\n");
717              unregister_filesystem(&shmem_fs_type);
718              return PTR_ERR(res);
719         }
720
721         devfs_mk_dir (NULL, "shm", NULL);
722         return 0;
723    }
```

先通过 kern_mount()安装特殊文件系统 shm。然后，如果在编译内核时选择了支持 devfs，就在/dev 目录（这是 devfs 的安装点）下建立一个子目录 shm，作为所有共享内存区文件的目录。我们在特殊文件/proc 一节中讲过，kern_mount()只是为一个特殊文件系统建立起作为一个已安装文件系统所需的所有数据结构，包括超级块以及 dentry 结构、inode 结构、还有 vfsmount 结构，并使 file_system_type 结构（在这里是 shmem_fs_type）中的指针 kern_mnt 指向其 vfsmount 结构。但是却并没有真的将这个文件系统“安装”在某个安装点上。换言之，并没有使这个特殊文件系统落实到哪一个有形的文件系统，即外部设备上。对于像管道一类无形又无名的文件系统这是没有问题的，因为管道文件实际上只存在于内存中，本来就不需要落实到外部设备上。可是，对于/proc 文件系统就得要在 kern_mount()以后再具体安装一次，将其安装在/proc 节点上，这样才能使其变成有名而可以通过路径名寻访。所以像/proc 那样的特殊文件系统的 FS_SINGLE 标志位为 1。用于共享内存区的文件系统 shm 则又不同了，这种文件是有形的，需要在文件中实际地存储数据，所以更是必须落实到某个物理的外设上才行。另一方面，共享内存区文件只对系统的当前运行有意义，一旦关机就失去了意义而不应继续存在，下一次安装时应该从空白开始，所以不适合放在普通的文件系统中。考虑到这个特点，显然最合适的是把 shm 落实到页面交换盘区中。下面读者就会看到，这一点已经实现在 shm 的驱动程序中。所以，shm 文件系统在 kern_mount()以后，不需要像/proc 文件系统那样再来安装一次，它的 FS_SINGLE 标志位为 0。至于它的 FS_LITTER 标志位为 1（见 702 行），则正是表示把这种文件系统拆卸下来时要丢弃其所有的资源（包括在/dev/devfsd 目录下的文件节点）。

这样，建立共享内存区的问题就转变成了在特殊文件系统“ shm”中建立映射文件的问题。这是由 shmem_file_setup()完成的，其代码在 mm/shmem.c 中：

==================== mm/shmem.c 800 853 ====================

[sys_shmget()>newseg()>shmem_file_setup()]

```
800    /*
801     * shmem_file_setup - get an unlinked file living in shmem fs
802     *
803     * @name: name for dentry (to be seen in /proc/<pid>/maps
804     * @size: size to be set for the file
805     *
806     */
807    struct file *shmem_file_setup(char * name, loff_t size)
808    {
809         int error;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct file *file;
    struct inode * inode;
    struct dentry *dentry, *root;
    struct qstr this;
    int vm_enough_memory(long pages);

    error = -ENOMEM;
    if (!vm_enough_memory((size) >> PAGE_SHIFT))
        goto out;

    this.name = name;
    this.len = strlen(name);
    this.hash = 0; /* will go */
    root = shmem_fs_type.kern_mnt->mnt_root;
    dentry = d_alloc(root, &this);
    if (!dentry)
        goto out;

    error = -ENFILE;
    file = get_empty_filp();
    if (!file)
        goto put_dentry;

    error = -ENOSPC;
    inode = shmem_get_inode(root->d_sb, S_IFREG | S_IRWXUGO, 0);
    if (!inode)
        goto close_file;

    d_instantiate(dentry, inode);
    dentry->d_inode->i_size = size;
    file->f_vfsmnt = mntget(shmem_fs_type.kern_mnt);
    file->f_dentry = dentry;
    file->f_op = &shmem_file_operations;
    file->f_mode = FMODE_WRITE | FMODE_READ;
    inode->i_nlink = 0;    /* It is unlinked */
    return(file);

close_file:
    put_filp(file);
put_dentry:
    dput (dentry);
out:
    return ERR_PTR(error);
}
```

前面讲过，每个共享内存区都有个区名，区名中包含着它的键值，这个区名就被用作文件名。我们把这段程序留给读者阅读。要说明的是，shmem_fs_type.kern_mnt->mnt_root 指向 shm 文件系统的 dentry 结构本身，所以文件建立在 shm 的根节点下。虽然这里在内存中建立起了文件的 dentry 结构和

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

inode 结构，这些数据结构的内容却不需要写到磁盘上去，因为一旦关机，这些数据结构的内容就失去意义了。这里的 842 行将指针 file->f_op 设置成指向 file_operations 结构 shmem_file_operations，而在返回到 newseg()中以后又在 212 行将其设置成指向 shm_file_operations，这两个数据结构都是只支持 mmap 操作的，但一个是 shmem_mmap()，而另一个是 shm_mmap()。两个数据结构分别定义在 mm/shmem.c 和 ipc/shm.c 中：

==================== mm/shmem.c 662 664 ====================

```
static struct file_operations shmem_file_operations = {
	mmap:		shmem_mmap
};
```

==================== ipc/shm.c 163 165 ====================

```
static struct file_operations shm_file_operations = {
	mmap:	shm_mmap
};
```

函数 shmem_file_setup()并不光是在 newseg()一处受到调用，还得照顾到从其它路径调用时的需要，所以才会在 shmem_file_setup()中把 file->f_op 设置成指向 shmem_file_operations，而在返回到 newseg()中以后又改成指向 shm_file_operations。函数 shm_mmap()和 shmem_mmap()的区别在于，前者所实现的是有形的 shm 文件，所以支持 open 和 close 操作；而后者所实现的是无形的 shm 文件，不能通过路径名来打开，所以不支持 open 和 close 操作。

此外，对 shm 节点的 inode 结构也有些特殊的设置，所以通过一个特殊的函数 shmem_get_inode()来分配和设置 inode 数据结构。其代码在 mm/shmem.c 中：

==================== mm/shmem.c 341 385 ====================

[sys_shmget()>newseg()>shmem_get_inode()]

```
struct inode *shmem_get_inode(struct super_block *sb, int mode, int dev)
{
	struct inode * inode;

	spin_lock (&sb->u.shmem_sb.stat_lock);
	if (!sb->u.shmem_sb.free_inodes) {
		spin_unlock (&sb->u.shmem_sb.stat_lock);
		return NULL;
	}
	sb->u.shmem_sb.free_inodes--;
	spin_unlock (&sb->u.shmem_sb.stat_lock);

	inode = new_inode(sb);
	if (inode) {
		inode->i_mode = mode;
		inode->i_uid = current->fsuid;
		inode->i_gid = current->fsgid;
		inode->i_blksize = PAGE_CACHE_SIZE;
		inode->i_blocks = 0;
		inode->i_rdev = to_kdev_t(dev);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            inode->i_mapping->a_ops = &shmem_aops;
            inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
            spin_lock_init (&inode->u.shmem_i.lock);
            switch (mode & S_IFMT) {
            default:
                init_special_inode(inode, mode, dev);
                break;
            case S_IFREG:
                inode->i_op = &shmem_inode_operations;
                inode->i_fop = &shmem_file_operations;
                break;
            case S_IFDIR:
                inode->i_op = &shmem_dir_inode_operations;
                inode->i_fop = &shmem_dir_operations;
                break;
            case S_IFLNK:
                inode->i_op = &page_symlink_inode_operations;
                break;
            }
            spin_lock (&shmem_ilock);
            list_add (&inode->u.shmem_i.list, &shmem_inodes);
            spin_unlock (&shmem_ilock);
        }
        return inode;
    }
```

这里有几点特殊之处。首先是参数 dev 为 0，因而 inode 结构中的 i_rdev 也是 0，因为这个 inode 结构并没有相应的磁盘上索引节点。其次，这个 inode 结构的 i_mapping->a_ops 指向 address_space_operations 数据结构的 shmem_aops，定义于 mm/shmem.c 中：

==================== mm/shmem.c 658 660 ====================

```
static struct address_space_operations shmem_aops = {
    writepage: shmem_writepage
};
```

这个数据结构提供的共享内存区的页面换出操作。此外，根据节点的性质，inode 结构中的指针 i_op 和 i_fop 也分别设置成指向不同的 inode_operations 结构和 file_operations 结构。最后，内核中为 shm 文件系统提供了一个专用的 inode 结构队列 shmem_inodes，所有用于 shm 文件系统的 inode 结构都通过一个专用的队列头挂在这个队列中。所以不管在什么时候都能找到属于 shm 文件系统的所有 inode 结构。

回到 newseg()的代码中，用于共享内存区页面换出 / 换入的文件已经创建并打开。这个已打开文件有个 file 结构，但是却不像一般已打开文件的 file 结构那样属于某个特定的进程，也不像一般 file 结构那样代表一个具体文件的读 / 写上下文，因为对共享内存区的访问完全是随机的，没有上下文的概念。同时，对于这个 file 结构也不能像对一般已打开文件那样通过一个已打开文件号来访问，因为所谓已打开文件号是个局部进程的概念，而不是全局的概念。从代码中可以看到，指向这个 file 的指针就保存在具体共享内存区的 shmid_kernel 数据结构中，shmid_kernel 结构既然代表着一个共享内存区，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

当然也就代表着它所映射的文件。

为一个共享内存区分配了 shmid_kernel 数据结构以后，还要使它与全局的 ipc_ids 数据结构 shm_ids 挂上钩，所以在回到 newseg()中以后要通过 shm_addid()来建立起这种联系（ipc/shm.c）：

==================== ipc/shm.c 89 92 ====================

[sys_shmget()>newseg()>shm_addid()]

```
static inline int shm_addid(struct shmid_kernel *shp)
{
	return ipc_addid(&shm_ids, &shp->shm_perm, shm_ctlmni+1);
}
```

至于 ipc_addid()的代码，读者已经在前一节（报文队列）中读过。用于共享内存区的 shm_ids 与用于报文队列的 msg_ids 类型相同，而共享内存区的 shmid_kernel 结构中的第一个成分 shm_perm 也是个 kern_ipc_perm 结构，它的起始地址就是整个 shmid_kernel 结构的起始地址。此外，shm_ctlmni 也与 msg_ctlmni 一样，是个对 shm_ids 中数组大小的控制量。

至此，共享内存区的创建就基本完成了。函数 newseg()和 sys_shmget()中其余的代码比较简单，我们把它留给读者。

最后，sys_shmget()也和 sys_msgget()一样，返回所创建共享内存区的一体化标识号。那是由 shm_buildid()计算的。

==================== ipc/shm.c 55 56 ====================

```
#define shm_buildid(id, seq) \
	ipc_buildid(&shm_ids, id, seq)
```

读者已经在前一节（报文队列）中读过 ipc_buildid()的代码。

在一个共享内存区创建以后，参加进来共享的进程则使用其键值通过 findkey()找到作为 ipc_id 结构数组下标的标识号，然后将其换算成一个一体化的标识号，那就与 sys_msgget()的处理过程一样了。

6.7.2 库函数 shmat()——建立共享内存区的映射

通过 shmget()以给定键值创建了一个共享内存区，或者取得了已创建共享内存区的标识号以后，还要通过 shmat()将这个内存区映射到本进程的虚存空间，此外，一个已经映射的共享内存区间也可以通过 shmat()改变其映射。这都是由 sys_shmat()完成的，其代码在 ipc/shm.c 中。我们分段阅读：

==================== ipc/shm.c 553 591 ====================

```
/*
 * Fix shmaddr, allocate descriptor, map shm, add attach descriptor to lists.
 */
asmlinkage long sys_shmat (int shmid, char *shmaddr, int shmflg, ulong *raddr)
{
	struct shmid_kernel *shp;
	unsigned long addr;
	struct file * file;
	int    err;
```

```
562     unsigned long flags;
563     unsigned long prot;
564     unsigned long o_flags;
565     int acc_mode;
566     void *user_addr;
567
568     if (shmid < 0)
569          return -EINVAL;
570
571     if ((addr = (ulong)shmaddr)) {
572          if (addr & (SHMLBA-1)) {
573               if (shmflg & SHM_RND)
574                    addr &= ~(SHMLBA-1);      /* round down */
575               else
576                    return -EINVAL;
577          }
578          flags = MAP_SHARED | MAP_FIXED;
579     } else
580          flags = MAP_SHARED;
581
582     if (shmflg & SHM_RDONLY) {
583          prot = PROT_READ;
584          o_flags = O_RDONLY;
585          acc_mode = S_IRUGO;
586     } else {
587          prot = PROT_READ | PROT_WRITE;
588          o_flags = O_RDWR;
589          acc_mode = S_IRUGO | S_IWUGO;
590     }
591
```

参数 shmaddr 为当前进程所要求映射的目标地址，也就是映射后该共享内存区在这个进程的用户空间中的起始地址。这个地址一般应该能被一个常数 SHMLBA 整除，该常数在 include/asm-i386/shmparam.h 中定义为 PAGE_SIZE，所以实际上这意味着与页面的边界对齐。如果调用参数 shmaddr 不能被 SHMLBA 整除，就要将 shmflg 中的 SHM_RND 标志设成 1，这样 sys_shmat()就会自动将此地址加以调整（见 573～574 行）。参数 shmaddr 也可以是 0，内核会根据当前进程的虚存空间使用情况为其分配一个。从代码中可以看出，在这两种情况下，将 flags 中的 MAP_FIXED 标志位分别设置成 1 和 0，以后将根据这个标志位作相应的处理。此外，还要将参数 shmflg 中的 SHM_RDONLY 标志位转换成若干用于内存映射和文件访问的标志，因为对共享内存区的管理涉及这两个方面。我们继续往下看：

==================== ipc/shm.c 592 606 ====================

[sys_shmat()]

```
592     /*
593      * We cannot rely on the fs check since SYSV IPC does have an
594      * aditional creator id...
595      */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    shp = shm_lock(shmid);
    if(shp == NULL)
        return -EINVAL;
    if (ipcperms(&shp->shm_perm, acc_mode)) {
        shm_unlock(shmid);
        return -EACCES;
    }
    file = shp->shm_file;
    shp->shm_nattch++;
    shm_unlock(shmid);

```

参数 shmid 为共享内存区的一体化标识号，通过 shm_lock()就可以找到代表这个共享内存区的 shmid_kernel 数据结构，并且加锁。可想而知，shm_lock()与前一节中的 msg_lock()应该很相似。事实上，两者确实是一样的：

==================== ipc/shm.c 50 50 ====================

```
#define shm_lock(id)   ((struct shmid_kernel*)ipc_lock(&shm_ids,id))
```

找到了目标共享区并将其锁住以后，就来检查访问权限，函数 ipcperms()的代码见前一节。共享内存区访问权限的管理与文件系统访问权限的管理相似，读者可参阅第 5 章中有关的内容。通过了对访问权限的检验，就进入实质性的阶段了：

==================== ipc/shm.c 607 627 ====================

[sys_shmat()]

```
    down(&current->mm->mmap_sem);
    user_addr = (void *) do_mmap (file, addr, file->f_dentry->d_inode->i_size, prot, flags, 0);
    up(&current->mm->mmap_sem);

    down (&shm_ids.sem);
    if(!(shp = shm_lock(shmid)))
        BUG();
    shp->shm_nattch--;
    if(shp->shm_nattch == 0 &&
      shp->shm_flags & SHM_DEST)
        shm_destroy (shp);
    shm_unlock(shmid);
    up (&shm_ids.sem);

    *raddr = (unsigned long) user_addr;
    err = 0;
    if (IS_ERR(user_addr))
        err = PTR_ERR(user_addr);
    return err;

}
```

从代码中可见，实质性的操作就是通过 do_mmap()建立起文件与虚存空间的映射。这个函数的代

码已经在第 2 章中阅读过了，读者不妨回过去复习一下。不过，在 do_mmap()的执行过程中有一些根据具体条件执行的操作，有必要结合建立共享内存区这么一个具体的情景再把这个过程走一遍，并作些补充的说明。

在 do_mmap()中，先对文件和区间两方面都作一些检查，包括起始地址与长度、已经映射的次数等等，然后进一步调用 do_mmap_pgoff()建立映射。

==================== mm/mmap.c 188 190 ====================

[sys_shmat()>do_mmap()>do_mmap_pgoff()]

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len,
	unsigned long prot, unsigned long flags, unsigned long pgoff)
{
. . . . . .
/* 对访问权限等等的检查 */
. . . . . .
```

==================== mm/mmap.c 250 260 ====================

```
	/* Obtain the address to map to. we verify (or select) it and ensure
	 * that it represents a valid section of the address space.
	 */
	if (flags & MAP_FIXED) {
		if (addr & ~PAGE_MASK)
			return -EINVAL;
	} else {
		addr = get_unmapped_area(addr, len);
		if (!addr)
			return -ENOMEM;
	}
```

前面讲过，如果在调用 shmat()时的参数 shmaddr 为 0，则在 sys_shmat()中将 flags 中的标志位 MAP_FIXED 设成 0，表示应由内核给分配一个。现在就通过 get_unmapped_area()来做这件事了。函数 get_unmapped_area()的代码在 mm/mmap.c 中：

==================== mm/mmap.c 374 398 ====================

[sys_shmat()>do_mmap()>do_mmap_pgoff()>get_unmapped_area()]

```
/* Get an address range which is currently unmapped.
 * For mmap() without MAP_FIXED and shmat() with addr=0.
 * Return value 0 means ENOMEM.
 */
#ifndef HAVE_ARCH_UNMAPPED_AREA
unsigned long get_unmapped_area(unsigned long addr, unsigned long len)
{
	struct vm_area_struct * vmm;

	if (len > TASK_SIZE)
		return 0;
	if (!addr)
		addr = TASK_UNMAPPED_BASE;
	addr = PAGE_ALIGN(addr);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        for (vmm = find_vma(current->mm, addr); ; vmm = vmm->vm_next) {
            /* At this point:  (!vmm || addr < vmm->vm_end). */
            if (TASK_SIZE - len < addr)
                return 0;
            if (!vmm || addr + len <= vmm->vm_start)
                return addr;
            addr = vmm->vm_end;
        }
    }
    #endif
```

读者自行阅读这段程序应该不会有困难。这里的常数 TASK_UNMAPPED_BASE 是在 include/asm-i386/processor.h 中定义的：

==================== include/asm-i386/processor.h 263 266 ====================

```
    /* This decides where the kernel will search for a free chunk of vm
     * space during mmap's.
     */
    #define TASK_UNMAPPED_BASE  (TASK_SIZE / 3)
```

也就是说，当给定的目标地址为 0 时，内核从(TASK_SIZE/3)即 1GB 处开始向上在当前进程的虚存空间中寻找一块足以容纳给定长度的区间。而当给定的目标地址不为 0 时，则从给定的地址开始向上寻找。函数 find_vma()在当前进程已经映射的虚存空间中找到第一个满足 vma->vm_end，即大于给定地址的区间。如果找不到这么一个区间。那就说明给定的地址尚未映射，因而可以使用；或者，如果所找到区间的起始地址高于给定地址加给定区间长度，那就说明在所找到区间之下有个足够大的空洞，因此也可以使用给定的地址。

至此，只要返回的地址非 0，addr 就已经是一个符合各种要求的虚存地址了。我们回到 do_mmap_pgoff()中继续往下看（mm/mmap.c）：

==================== mm/mmap.c 262 306 ====================

[sys_shmat()>do_mmap()>do_mmap_pgoff()]

```
        /* Determine the object being mapped and call the appropriate
         * specific mapper. the address has already been validated, but
         * not unmapped, but the maps are removed from the list.
         */
        vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
        if (!vma)
            return -ENOMEM;

        vma->vm_mm = mm;
        vma->vm_start = addr;
        vma->vm_end = addr + len;
        vma->vm_flags = vm_flags(prot,flags) | mm->def_flags;

        if (file) {
            VM_ClearReadHint(vma);
            vma->vm_raend = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

        if (file->f_mode & FMODE_READ)
            vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
        if (flags & MAP_SHARED) {
            vma->vm_flags |= VM_SHARED | VM_MAYSHARE;

            /* This looks strange, but when we don't have the file open
             * for writing, we can demote the shared mapping to a simpler
             * private mapping. That also takes care of a security hole
             * with ptrace() writing to a shared mapping without write
             * permissions.
             *
             * We leave the VM_MAYSHARE bit on, just to get correct output
             * from /proc/xxx/maps..
             */
            if (!(file->f_mode & FMODE_WRITE))
                vma->vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
        }
    } else {
        vma->vm_flags |= VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
        if (flags & MAP_SHARED)
            vma->vm_flags |= VM_SHARED | VM_MAYSHARE;
    }
    vma->vm_page_prot = protection_map[vma->vm_flags & 0x0f];
    vma->vm_ops = NULL;
    vma->vm_pgoff = pgoff;
    vma->vm_file = NULL;
    vma->vm_private_data = NULL;

```

每个虚存区间都要有个 vm_area_struct 数据结构，所以通过 kmem_cache_alloc()为待映射的区间分配一个，并加以设置。如果调用 do_mmap_pgoff()时的 file 结构指针为 0，则目的仅在于创建虚存区间，或者说仅在于建立从物理空间到虚存区间的映射。而如果目的在于建立从文件到虚存区间的映射，那就要把为文件设置的访问权限考察进去。至此，代表着我们所需虚存区间的数据结构已经创建了（不过尚未插入代表着当前进程虚存空间的 mm_struct 结构中）。可是，在某些条件下却还不得不将它撤销。为什么呢？我们继续往下看：

==================== mm/mmap.c 307 322 ====================

[sys_shmat()>do_mmap()>do_mmap_pgoff()]

```
    /* Clear old maps */
    error = -ENOMEM;
    if (do_munmap(mm, addr, len))
        goto free_vma;

    /* Check against address space limit. */
    if ((mm->total_vm << PAGE_SHIFT) + len
       > current->rlim[RLIMIT_AS].rlim_cur)
        goto free_vma;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
316
317     /* Private writable mapping? Check memory availability.. */
318     if ((vma->vm_flags & (VM_SHARED | VM_WRITE)) == VM_WRITE &&
319         !(flags & MAP_NORESERVE)                 &&
320         !vm_enough_memory(len >> PAGE_SHIFT))
321          goto free_vma;
322
```

在什么条件下要把已经创建（但是尚未生效）的 vm_area_struct 数据结构撤销呢？首先，这里调用了一个函数 do_munmap()。它检查目标地址在当前进程的虚存空间是否已经在使用，如果已经在使用，就要将老的映射去除，将老的区间释放。要是这个操作失败，那当然不能重复映射同一个目标地址，所以就得转移到 free_vma，把已经分配的 vm_area_struct 数据结构撤销。函数 do_munmap()的代码在 mm/mmap.c 中，读者已经在第 2 章中读过它的代码。也许读者会感到奇怪，这个区间不是在前面调用 get_unmapped_area()找到的吗？怎么可能会原来就已有映射了呢？回过头去注意看一下就可知道，那只是当调用参数 shmaddr 为 0 时的情况，而当 shmaddr 不为 0 时则尚未对此加以检查。

除此之外，还有两个情况也会导致撤销已经分配的 vm_area_struct 数据结构。一个是如果当前进程对虚存空间的使用超出了为其设置的上限（313～315 行）。另一个是在要求建立由当前进程专用的可写区间，而物理页面的数量已经（暂时）不足（318～321 行）。

读者也许还要问：为什么不把对所有条件的检验放在分配 vm_area_struct 数据结构之前呢？问题在于，在通过 kmem_cache_alloc()分配 vm_area_struct 数据结构的过程中，有可能会发生供这种数据结构专用的 slab 已经用完，而不得不补充分配更多物理页面的情况。而分配物理页面的过程，则又有可能因一时不能满足要求而只好先调度别的进程运行。这样，当前进程从 kmem_cache_alloc()返回时，可能已经有别的进程或线程，特别是由本进程 clone()出来的线程运行过了，所以就不能排除这些条件已经改变的可能。所以，读者在内核中常常能看到先分配某项资源，然后检测条件，如果条件不符再将资源释放（而不是先检测条件，后分配资源）的情景。关键就在于分配资源的过程中是否有可能发生调度，以及其它进程或线程的运行有否可能改变这些条件。以这里的第三个条件为例，如果发生过调度，那就显然是可能改变的。

继续往下看 do_mmap_pgoff()的代码（mm/mmap.c）：

==================== mm/mmap.c 323 340 ====================

[sys_shmat()>do_mmap()>do_mmap_pgoff()]

```
323     if (file) {
324          if (vma->vm_flags & VM_DENYWRITE) {
325               error = deny_write_access(file);
326               if (error)
327                    goto free_vma;
328               correct_wcount = 1;
329          }
330          vma->vm_file = file;
331          get_file(file);
332          error = file->f_op->mmap(file, vma);
333          if (error)
334               goto unmap_and_free_vma;
335     } else if (flags & MAP_SHARED) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        error = shmem_zero_setup(vma);
        if (error)
            goto free_vma;
    }
```

如果要建立的是从文件到虚存区间的映射，而在调用 do_mmap()时的参数 flags 中的 MAP_DENYWRITE 标志位为 1（这个标志位在前面 273 行引用的宏操作 vm_flags()中转换成 VM_DENYWRITE），那就表示不允许通过常规的文件操作对此文件进行写访问，所以要调用 deny_write_access()排斥常规的文件操作，详见“文件系统”一章中的有关内容。对于 shm 文件，读者可以在 sys_shmat()的代码中看出这个标志位一定是 0，所以不存在这个问题。这是因为对 shm 文件本来就不允许按常规的可写文件打开。

函数 get_file()的作用只是递增 file 结构中的共享计数。

每种文件系统都有个 file_operations 数据结构，其中的函数指针 mmap 提供了建立从该类文件到虚存区间的映射的操作。我们在前面已经看到过 shm 文件系统的 file_operations 数据结构，它只提供一种文件操作，那就是 mmap，具体的函数是 shm_mmap()。其代码在 ipc/shm.c 中：

==================== ipc/shm.c 155 161 ====================
[sys_shmat()>do_mmap()>do_mmap_pgoff()>shm_mmap()]

```
155 static int shm_mmap(struct file * file, struct vm_area_struct * vma)
156 {
157     UPDATE_ATIME(file->f_dentry->d_inode);
158     vma->vm_ops = &shm_vm_ops;
159     shm_inc(file->f_dentry->d_inode->i_ino);
160     return 0;
161 }
```

这个函数非常简单，实质性的操作其实只有一行，那就是 158 行将虚存区间控制结构中的指针 vm_ops 设置成指向数据结构 shm_vm_ops。这个结构的定义为：

==================== ipc/shm.c 167 171 ====================

```
static struct vm_operations_struct shm_vm_ops = {
    open:    shm_open,/* callback for a new vm-area open */
    close:   shm_close,    /* callback for when the vm-area is released */
    nopage:  shmem_nopage,
};
```

读者也许感到困惑，在文件与虚存区间之间建立映射难道就这么简单？其实，具体的映射是非常动态、经常在变的。所谓文件与虚存区之间的映射包含着两个环节，一是物理页面与文件映象之间的换入／换出，二是物理页面与虚存页面之间的映射。这二者都是很动态的，所以，重要的并不是建立起一个特定的映射，而是建立起一套机制，使得一旦需要时就可以根据当时的具体情况建立起新的映射。另一方面，在计算机技术中有一个称为“lazy computation”的概念，就是说有些为将来作某种准备而进行的操作（计算）应该推迟到真正需要时才进行。这是因为实际运行中的情况千变万化，有时候花了老大的劲才完成了准备，实际上却根本没有用到，或者只用到了很小一部分，从而造成了浪费。就以这里共享内存区的映射来说，也许共享区的大小是 100 个页面，而实际上在相当长的时间里只用

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

到了其中的一个页面，而映射 99 个页面的开销都不是可以忽略不计的。何况，长期不用的页面还得费劲把它们换出哩。考虑到这些因素，还不如到真正需要用到一个页面时再来建立其映射，用到几个页面就映射几个页面。当然，那样很可能会因为分散处理而使具体映射每一个页面的开销增加，所以这里有个权衡利弊的问题，具体的决定往往要建立在统计的基础上。这里，对于共享内存区的映射正是运用了这个概念，把具体页面的映射推迟到了真正需要的时候才进行。具体地，就是为物理页面的换入（以及为映射的建立而准备的下一个函数，这就是 shm_nopage()）。等一下我们还会回到这个话题上来。

回到 do_mmap_pgoff()的代码中，把共享内存区的 vm_area_struct 结构插入当前进程的虚存空间，就完成了共享内存区映射机制的建立，虽然具体页面的映射都尚未建立。

回到 sys_shmat()的代码。最后通过参数 raddr 返回实际的映射地址。至此，sys_shmat()的操作就基本完成了，给定的共享内存区已经纳入了当前进程的存储空间。

如前所述，在 sys_shmat()中实际上并没有建立页面的映射，而是把它推迟到了实际需要的时候。所以，在将一块共享内存区纳入一个进程的存储空间以后，当其中的任何一个页面首次受到访问时就会因为“缺页”而产生一次页面异常。读者不妨回到第 2 章中，从 do_page_fault()开始，顺着 handle_mm_fault()、handle_pte_fault()，一直到 do_no_page()。在 do_no_page()中，如果产生异常的地址所属区间的指针 vm_ops 指向一个 vm_operations_struct 数据结构，并且该结构中的函数指针 nopage 非零，就会调用这个函数来建立所在页面的映射表项。下面是 do_no_page()中的一个片段：

==================== mm/memory.c 1097 1105 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()]

```
1097      if (!vma->vm_ops || !vma->vm_ops->nopage)
1098           return do_anonymous_page(mm, vma, page_table, write_access, address);
1099
1100      /*
1101       * The third argument is "no_share", which tells the low-level code
1102       * to copy, not share the page even if sharing is possible.  It's
1103       * essentially an early COW detection.
1104       */
1105      new_page = vma->vm_ops->nopage(vma, address & PAGE_MASK, (vma->vm_flags &
VM_SHARED)?0:write_access);
     . . . . . .
```

==================== mm/memory.c 1123 1123 ====================

```
1123      entry = mk_pte(new_page, vma->vm_page_prot);
     . . . . . .
```

==================== mm/memory.c 1129 1129 ====================

```
1129      set_pte(page_table, entry);
```

读者在前面已经看到，对于共享内存区内的页面，这个指针指向 shmem_nopage()。其代码在 mm/shmem.c 中，我们又得分段来看：

==================== mm/shmem.c 237 271 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()>shmem_nopage()]

```
237  /*
238   * shmem_nopage - either get the page from swap or allocate a new one
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
239   *
240   * If we allocate a new one we do not mark it dirty. That's up to the
241   * vm. If we swap it in we mark it dirty since we also free the swap
242   * entry since a page cannot live in both the swap and page cache
243   */
244  struct page * shmem_nopage(struct vm_area_struct * vma, unsigned long address, int no_share)
245  {
246       unsigned long size;
247       struct page * page;
248       unsigned int idx;
249       swp_entry_t *entry;
250       struct inode * inode = vma->vm_file->f_dentry->d_inode;
251       struct address_space * mapping = inode->i_mapping;
252       struct shmem_inode_info *info;
253  
254       idx = (address - vma->vm_start) >> PAGE_SHIFT;
255       idx += vma->vm_pgoff;
256  
257       down (&inode->i_sem);
258       size = (inode->i_size + PAGE_CACHE_SIZE - 1) >> PAGE_CACHE_SHIFT;
259       page = NOPAGE_SIGBUS;
260       if ((idx >= size) && (vma->vm_mm == current->mm))
261            goto out;
262  
263       /* retry, we may have slept */
264       page = __find_lock_page(mapping, idx, page_hash (mapping, idx));
265       if (page)
266            goto cached_page;
267  
268       info = &inode->u.shmem_i;
269       entry = shmem_swp_entry (info, idx);
270       if (!entry)
271            goto oom;
```

先计算出页面在所映射文件，即共享内存区内的页面号 idx。如果某个（进程的）特定虚存区间与整个共享内存区之间有位移，就要把位移也考虑进去（255 行）。算出页面号以后，就可以通过__find_lock_page()在杂凑表队列中寻找该页面的 page 结构。如果找到了，那就说明这个页面已经在内存中的页面缓冲队列里，只要重建映射就可以了。否则，就要通过 shmem_swp_entry()进一步确定这个页面是从未映射，还是已经换出到交换盘区上。这个函数的代码在 mm/shmem.c 中，这是了解共享内存区内页面换出／换入操作的关键。

==================== mm/shmem.c 52 72 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()>shmem_nopagc()>shmem_swp_entry()]

```
static swp_entry_t * shmem_swp_entry (struct shmem_inode_info *info, unsigned long index)
{
     if (index < SHMEM_NR_DIRECT)
          return info->i_direct+index;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        index -= SHMEM_NR_DIRECT;
        if (index >= ENTRIES_PER_PAGE*ENTRIES_PER_PAGE)
             return NULL;

        if (!info->i_indirect) {
             info->i_indirect = (swp_entry_t **) get_zeroed_page(GFP_USER);
             if (!info->i_indirect)
                  return NULL;
        }
        if(!(info->i_indirect[index/ENTRIES_PER_PAGE])) {
             info->i_indirect[index/ENTRIES_PER_PAGE] = (swp_entry_t *) get_zeroed_page(GFP_USER);
             if (!info->i_indirect[index/ENTRIES_PER_PAGE])
                  return NULL;
        }

        return info->i_indirect[index/ENTRIES_PER_PAGE]+index%ENTRIES_PER_PAGE;
   }
```

读者在“文件系统”一章中已经知道，inode 结构中有个 union，根据文件系统的不同而解释成不同的数据结构。对于 shm 文件系统，这个 union 解释成 shmem_inode_info 数据结构，定义于 include/linux/shmem_fs.h 中：

==================== include/linux/shmem_fs.h 16 27 ====================

```
typedef struct {
     unsigned long val;
} swp_entry_t;

struct shmem_inode_info {
     spinlock_t     lock;
     swp_entry_t    i_direct[SHMEM_NR_DIRECT]; /* for the first blocks */
     swp_entry_t   **i_indirect; /* doubly indirect blocks */
     unsigned long swapped;
     int       locked;    /* into memory */
     struct list_head   list;
};
```

结构中引用的类型 swp_entry_t 实际上就是 32 位无符号整数，若为非 0 就表示一个页面在交换设备上的页面号。数组 i_direct[]代表着一个共享内存区的开头 16 个页面，即 64K 字节（SHMEM_NR_DIRECT 定义为 16）。对于一般的共享内存区，这个大小已经够了。如果不够，就通过指针 i_indirect 引入间接映射；这个指针指向一个用作指针数组的页面。页面中是 1024 个指针，每个指针又指向另一个用作 swp_entry_t 数组的页面，每个页面中可以容纳 1024 项 swp_entry_t。这样，总的潜在的容量是 1M 个页面，即 4G 字节！显然，这种机制与 Ext2 文件系统的多重间接映射相似，但要简单一些。原因是这些数据结构和数组不需要存储在磁盘上。函数 shmem_swp_entry()返回指向目标 swp_entry_t 项的指针。这个指针不应为 0。为 0 时就说明分配不到用于间接映射的页面，所以转向 oom 作“Out-Of-Memory”出错处理。只要这个指针有效，根据 swp_entry_t 的内容就可以判断页面是否在交换设备上。我们继续往下看：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== mm/shmem.c 272 297 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()>shmem_nopage()]

```
272         if (entry->val) {
273                 unsigned long flags;
274
275                 /* Look it up and read it in.. */
276                 page = lookup_swap_cache(*entry);
277                 if (!page) {
278                         lock_kernel();
279                         swapin_readahead(*entry);
280                         page = read_swap_cache(*entry);
281                         unlock_kernel();
282                         if (!page)
283                                 goto oom;
284                 }
285
286                 /* We have to this with page locked to prevent races */
287                 spin_lock (&info->lock);
288                 swap_free(*entry);
289                 lock_page(page);
290                 delete_from_swap_cache_nolock(page);
291                 *entry = (swp_entry_t) {0};
292                 flags = page->flags & ~((1 << PG_uptodate) | (1 << PG_error) | (1 << PG_referenced) | (1 <<
PG_arch_1));
293                 page->flags = flags | (1 << PG_dirty);
294                 add_to_page_cache_locked(page, mapping, idx);
295                 info->swapped--;
296                 spin_unlock (&info->lock);
297         } else {
```

如果目标 swp_entry_t 项的值为非 0，就表示目标页面在交换设备上，所以要把页面换入。对于认真阅读过第 2 章中有关换出 / 换入操作的读者，余下的代码已经不难理解了。不过要注意 291 行，在把目标页面读入内存以后，这里把页面的 swp_entry_t 项清 0，就是说，一个页面的 swp_entry_t 项为 0，既可以表示页面从未映射，也可以表示该页面在内存中。这样做并无不妥，因为如果页面在内存中就一定在缓冲队列中，前面的__find_lock_page()就应成功，或者根本就不会发生缺页异常，而不会到达上面的 268 行了。

我们继续往下看 swp_entry_t 的值为 0 时的情景：

==================== mm/shmem.c 297 339 ====================

[do_page_fault()>handle_mm_fault()>handle_pte_fault()>do_no_page()>shmem_nopage()]

```
297         } else {
298                 spin_lock (&inode->i_sb->u.shmem_sb.stat_lock);
299                 if (inode->i_sb->u.shmem_sb.free_blocks == 0)
300                         goto no_space;
301                 inode->i_sb->u.shmem_sb.free_blocks--;
302                 spin_unlock (&inode->i_sb->u.shmem_sb.stat_lock);
303                 /* Ok, get a new page */
304                 page = page_cache_alloc();
```

```
305             if (!page)
306                 goto oom;
307             clear_user_highpage(page, address);
308             inode->i_blocks++;
309             add_to_page_cache (page, mapping, idx);
310         }
311         /* We have the page */
312         SetPageUptodate (page);
313 
314     cached_page:
315         UnlockPage (page);
316         up(&inode->i_sem);
317 
318         if (no_share) {
319             struct page *new_page = page_cache_alloc();
320 
321             if (new_page) {
322                 copy_user_highpage(new_page, page, address);
323                 flush_page_to_ram(new_page);
324             } else
325                 new_page = NOPAGE_OOM;
326             page_cache_release(page);
327             return new_page;
328         }
329 
330         flush_page_to_ram (page);
331         return(page);
332     no_space:
333         spin_unlock (&inode->i_sb->u.shmem_sb.stat_lock);
334     oom:
335         page = NOPAGE_OOM;
336     out:
337         up(&inode->i_sem);
338         return page;
339     }
```

既然执行到了 272 行，而目标 swp_entry_t 项的值又为 0，那就只有一个解释，就是目标页面从未映射过。所以，这里通过 page_cache_alloc()分配一个空闲内存页面，并通过 clear_user_highpage()将该页面清成全 0，再将其链入缓冲页面队列中。

至此，当执行到标号 cached_page 处时，我们要么从缓冲页面队列里找到了目标页面，要么已经新分配了一个空闲页面并已将其链入缓冲页面队列。此时还要考虑到一个特殊情况，那就是如果调用 shmem_nopage()时的参数 no_share 非 0（见 do_no_page()中的 1105 行），则表示页面所在的虚存区间实际上不允许共享，此时需要通过 copy_user_highpage()为目标页面，复制一份由当前进程专用的副本。最后，将指向目标页面 page 结构或其副本的指针返回给 do_no_page()，由 do_no_page()在当前进程的页面映射表中建立起映射。

还要说明一点特殊之处。读者在第 2 章中看到，对于普通的存储空间映射，当一个页面的映射断

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

开时，即物理页面不在内存时，相应页面映射表项中的 P 标志位为 0，但是整个表项并不是 0，此时的表项指明了页面的去向。但是，对于通过 do_mmap()建立的文件映射，却不需要由页面映射表中的表项来指明页面的去向，所以当页面不在内存中时表项的内容为全 0（参看第 2 章中 try_to_swap_out()的有关代码，特别是 83，125，135 以及 101～107 行）。对于普通文件的映射，文件的 inode 结构中已经提供了所需的全部信息，包括文件的内容存储在哪一个设备上，目标页面又映射到设备上的哪几个记录块。对于共享内存区文件，其 inode 结构同样也提供了所需的全部信息，只不过文件的内容总是存储在交换设备上，并且映射的方式也略有不同而已。

建立起对共享内存区页面的映射以后，有关的进程就可以像对一般存储页面一样地读写。每当访问一个页面时，CPU 中的硬件保证了将相应页面表项中的_PAGE_ACCESSED 标志位设成 1，如果是写访问则还要将_PAGE_DIRTY 标志位也设成 1。通过检查页面表项中的这些标志位，就可以知道参与共享的各个进程是否访问了这个页面，以及是否写访问。

再来看页面的换出。在第 2 章中，读者看到内核线程 kswapd()通过一个函数 page_launder()扫描 inactive_dirty_list 队列，将已经受到过写访问，但是已经不活跃的页面写到交换设备或者文件中去。除 kswapd()以外，另一个内核线程 bdflush()也会调用 page_launder()，还有设备驱动层的 block_write()，block_read()，bread()以及 ext2_getblk()等函数也会通过 getblk()和 refill_freelist()，辗转地调用 page_launder()。所以，page_launder()受到调用的机会是很多的。下面是 page_launder()中的一个片段：

```
==================== mm/vmscan.c 537 547 ====================
[kswapd()>do_try_to_free_pages()>page_launder()]
            /*
             * Dirty swap-cache page? Write it out if
             * last copy..
             */
            if (PageDirty(page)) {
                int (*writepage)(struct page *) = page->mapping->a_ops->writepage;
                int result;

                if (!writepage)
                    goto page_active;

    . . . . . .
==================== mm/vmscan.c 561 561 ====================
                result = writepage(page);
    . . . . . .
==================== mm/vmscan.c 571 571 ====================
            }
```

对于共享内存区的页面，写入的目标是文件，但是这个文件在交换设备上。读者在第 2 章中看到，页面的 page 数据结构中有个指针 mapping，指向一个 address_space 数据结构，这个数据结构通常就在页面所属文件的 inode 结构内部。在 address_space 结构里面有三个队列头，即 clean_pages，dirty_pages 以及 locked_pages，属于同一文件的“脏”页面都挂在这个文件的 dirty_pages 队列中。同时，address_space

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构里面又有个指针 a_ops，指向所属文件系统的 address_space_operations，这是在创建具体文件的 inode 结构时设置好了的。对于 shm 文件系统，这个数据结构是 shmem_aops，它提供了 shm 文件系统的 writepage 函数 shmem_writepage()，其代码在 mm/shmem.c 中：

==================== mm/shmem.c 194 235 ====================

[kswapd()>do_try_to_free_pages()>page_launder()>shmem_writepage()]

```
/*
 * Move the page from the page cache to the swap cache
 */
static int shmem_writepage(struct page * page)
{
     int error;
     struct shmem_inode_info *info;
     swp_entry_t *entry, swap;

     info = &page->mapping->host->u.shmem_i;
     if (info->locked)
          return 1;
     swap = __get_swap_page(2);
     if (!swap.val)
          return 1;

     spin_lock(&info->lock);
     entry = shmem_swp_entry (info, page->index);
     if (!entry)   /* this had been allocted on page allocation */
          BUG();
     error = -EAGAIN;
     if (entry->val) {
                __swap_free(swap, 2);
          goto out;
        }

        *entry = swap;
     error = 0;
     /* Remove the from the page cache */
     lru_cache_del(page);
     remove_inode_page(page);

     /* Add it to the swap cache */
     add_to_swap_cache(page, swap);
     page_cache_release(page);
     set_page_dirty(page);
     info->swapped++;
out:
     spin_unlock(&info->lock);
     UnlockPage(page);
     return error;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里通过__get_swap_page()从交换设备上分配一个页面。注意这里的调用参数为 2，表示应将所分配的页面的使用计数设置成 2，而不是要分配两个页面。接着，根据物理页面号，通过 shmem_swp_entry()在文件的 swp_entry_t 表中找到相应的表项。如果表项的内容为非 0，就表示这个页面在交换设备上已经有对应页面了，所以此时要通过__swap_free()归还刚才分配到的盘上页面。

然后，把页面从 LRU 队列中脱链，转移到交换设备的换出队列 swapper_space 中，并且将其 page 结构中的指针 mapping 设置成指向 swapper_space。这么一来，当 page_launder()再次扫描到这个页面时，它的 writepage 函数就变成由 swapper_space 通过 swap_aops 提供的 swap_writepage()了。此后的操作就与普通页面的换出完全相同了，因为这些页面同样是以交换设备为目标的。为方便读者阅读，我们在这里列出两个有关的函数：

==================== mm/swap_state.c 20 24 ====================
[kswapd()>do_try_to_free_pages()>page_launder()>swap_writepage()]

```
20   static int swap_writepage(struct page *page)
21   {
22        rw_swap_page(WRITE, page, 0);
23        return 0;
24   }
```

==================== mm/swap_state.c 20 24 ====================
[kswapd()>do_try_to_free_pages()>page_launder()>swap_writepage()>rw_swap_page()]

```
107  void rw_swap_page(int rw, struct page *page, int wait)
108  {
109       swp_entry_t entry;
110
111       entry.val = page->index;
112
113       if (!PageLocked(page))
114            PAGE_BUG(page);
115       if (!PageSwapCache(page))
116            PAGE_BUG(page);
117       if (page->mapping != &swapper_space)
118            PAGE_BUG(page);
119       if (!rw_swap_page_base(rw, entry, page, wait))
120            UnlockPage(page);
121  }
```

再往下就是设备驱动的事了。

读者大概注意到了，shm_vm_ops 中还有其它两个指针 open 与 close 也都是有定义的。这是为什么呢？我们在 sys_shmget()和 sys_shmat()的代码中都没有看到调用它的 open 操作啊？答案是，这是在 fork()一个进程的过程中使用的（ipc/shm.c）：

==================== ipc/shm.c 107 111 ====================

```
107  /* This is called by fork, once for every shm attach. */
108  static void shm_open (struct vm_area_struct *shmd)
109  {
110       shm_inc (shmd->vm_file->f_dentry->d_inode->i_ino);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}

```

当 fork()一个进程时，要把父进程的每个 vm_area_struct 数据结构都复制到子进程中。对于代表着一个共享内存区的 vm_area_struct 来说，还要将它链入到所属共享内存区数据结构中的一个队列 attaches 中，这是与普通虚存区间不同的地方。所以要为之提供一个函数 shm_open()来完成此项操作。读者可以参阅第 4 章中有关 do_fork()的一节，顺着 do_fork()，copy_mm()和 dup_mmap()的路线，最后进入 dup_mmap()。在那里读者可以看到这么几行（kernel/fork.c）：

==================== kernel/fork.c 171 174 ====================
[sys_fork()>do_fork()>copy_mm()>dup_mmap()]

```
		/* Copy the pages, but defer checking for errors */
		retval = copy_page_range(mm, current->mm, tmp);
		if (!retval && tmp->vm_ops && tmp->vm_ops->open)
			tmp->vm_ops->open(tmp);
```

就是说，如果复制出来的 vm_area_struct 结构 tmp 的 vm_ops 指针指向一个 vm_operations_struct 数据结构，并且该结构中的指针 open 非零，就调用这个函数。在这里，就是调用 shm_open()，递增共享内存区所映射文件的共享计数。与此相类似，当一个进程 exit()时，要释放其所有的虚存区间。如果一个虚存区间实际上是一个共享区，就会调用 shm_close()递减这个计数。

6.7.3 库函数 shmdt()——撤销共享内存区的映射

一个进程在通过 SHMAT 操作与一个共享内存区挂上钩并建立起映射以后，就可以像对普通内存空间一样通过虚存地址访问这块空间。当不再需要这块空间时，可以通过系统调用 ipc()的 SHMDT 操作与之脱钩，具体由 sys_shmdt()完成。此外，一个刚 fork()出来的子进程在开始执行一个新的程序时也要释放从父进程复制下来的各个虚存区间。如果一个虚存区间实际代表着一个共享内存区的话，就会通过数据结构 shm_vm_ops 中的指针 close 调用 shm_close()，在那里也要执行类似的操作。

函数 sys_shmdt()的代码比较简单（ipc/shm.c）：

==================== ipc/shm.c 629 647 ====================

```
/*
 * detach and kill segment if marked destroyed.
 * The work is done in shm_close.
 */
asmlinkage long sys_shmdt (char *shmaddr)
{
	struct mm_struct *mm = current->mm;
	struct vm_area_struct *shmd, *shmdnext;

	down(&mm->mmap_sem);
	for (shmd = mm->mmap; shmd; shmd = shmdnext) {
		shmdnext = shmd->vm_next;
		if (shmd->vm_ops == &shm_vm_ops
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
               && shmd->vm_start - (shmd->vm_pgoff << PAGE_SHIFT) == (ulong) shmaddr)
                do_munmap(mm, shmd->vm_start, shmd->vm_end - shmd->vm_start);
        }
        up(&mm->mmap_sem);
        return 0;
}
```

其主体是 do_munmap()，我们已在第 2 章读过它的代码。以前讲过，每个 vm_area_struct 数据结构代表着一个独立的虚存区间，而属性不同的区间即使地址连续也分属不同的独立虚存区间，由不同的 vm_area_struct 数据结构代表。而 vm_operations_struct 数据结构为 shm_vm_ops，则正是用于共享内存区的独立区间的特征。另一方面，与一个共享内存区脱钩必须是与整个共享内存区脱钩，而不是它的一部分。所以共享内存区的起始地址必须与一个虚存区间的实际起始地址相符。

6.7.4 库函数 shmctl()——对共享内存区的控制与管理

像报文队列一样，对共享内存区也可以通过系统调用 ipc()的 SHMCTL 操作加以控制，或收集其状态及统计信息。函数 sys_shmctl()的逻辑很简单，而且所有 SysV IPC 机制在这方面基本上都一样。例如都是通过 IPC_RMID 命令撤消一项设施，所以我们不在这里列出其代码了，只是提一下其特殊之处。

对于一块共享内存区，可以在 SHMCTL 操作中分别通过 SHM_LOCK 和 SHM_UNLOCK 两条命令来加锁和去锁，也就是禁止和恢复该区间的各个页面换出或换入操作。显然，这要由共享内存和页面换出／换入两个机制相互配合才能实现。

通过 SHMCTL 操作对共享内存区加锁时，一方面，在共享内存区的 shmid_kernel 结构中把一个标志位 SHM_LOCKED 设成 1；另一方面，同时在相应 shm 文件的 inode 结构内部将其 shmem_inode_info 结构中的字段 locked 也设置成 1。读者应该还记得，对于 shm 文件，inode 结构中的 union 解释为 shmem_inode_info 结构。

而在 shmem_writepage()中，则要检查页面所属 shm 文件的 inode 结构内部的这个标志，如果为 1 就提前返回（见上面 shmem_writepage()代码中的 203～205 行），从而使对该页面的 writepage 操作变成了空操作。

6.8 信号量

前面讲过，共享内存为进程间通讯提供了一种效率很高的手段，但是这种机制所提供的只是狭义的“通信”手段，而并不提供进程间同步的机能。所以，共享内存作为广义的进程间通信手段还必须要有其它机制的配合。同时，除共享内存以外，还有其它需要共享资源的场合也需要有进程间同步的手段。举例来说，如果有两个进程共享同一个 tty 终端，并且各自都通过 printf()在屏幕上显示一些字符串，就很可能会使来自两个不同进程的字符在屏幕上混成一片，而令人无法阅读。所以，原则上讲只要两个进程直接共享某个资源，就得要有互相同步的手段。其中有些同步是由内核自动提供的（例如对 CPU，对管道机制中的缓冲区，等等），而有些则要由所涉及的进程自己来关心。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

由此可见，将已经在内核中使用的进程间同步机制“信号量”推广到用户空间，是很自然的事。SysV IPC 中的“信号量”机制就是这样一种推广，所以实际上应该称为“用户空间信号量”以示区别。不过，只要不至于引起混淆，我们在文中就还是称之为“信号量”，读者应注意区分用户空间信号量与以前讲过的内核信号量这二者之间的区别。同时，还要认识到“用户空间信号量”这种机制是由内核来支持，在系统空间中实现的，只不过是由用户进程直接使用而已。

与信号量有关的操作有三种，就是 SEMGT，SEMOP 以及 SEMCTL，分别由 sys_semget()，sys_semop() 和 sys_semctl()实现。有关的代码和定义基本都在文件 ipc/sem.c 和 include/linux/sem.h 中。

6.8.1 库函数 semget()——创建或寻找信号量

函数 sys_semget()的代码与 sys_msgget()几乎一模一样，主要的区别只是把 sys_msgget()中的子程序调用 newque()换成了 newary()。所以我们就来看看 newary()（ipc/sem.c）:

==================== ipc/sem.c 112 148 ====================

```
static int newary (key_t key, int nsems, int semflg)
{
	int id;
	struct sem_array *sma;
	int size;

	if (!nsems)
		return -EINVAL;
	if (used_sems + nsems > sc_semmns)
		return -ENOSPC;

	size = sizeof (*sma) + nsems * sizeof (struct sem);
	sma = (struct sem_array *) ipc_alloc(size);
	if (!sma) {
		return -ENOMEM;
	}
	memset (sma, 0, size);
	id = ipc_addid(&sem_ids, &sma->sem_perm, sc_semmni);
	if(id == -1) {
		ipc_free(sma, size);
		return -ENOSPC;
	}
	used_sems += nsems;

	sma->sem_perm.mode = (semflg & S_IRWXUGO);
	sma->sem_perm.key = key;

	sma->sem_base = (struct sem *) &sma[1];
	/* sma->sem_pending = NULL; */
	sma->sem_pending_last = &sma->sem_pending;
	/* sma->undo = NULL; */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
143         sma->sem_nsems = nsems;
144         sma->sem_ctime = CURRENT_TIME;
145         sem_unlock(id);
146 
147         return sem_buildid(id, sma->sem_perm.seq);
148   }
```

先看调用参数。只要回顾一下 sys_msgget()，这里面的参数 key 和 semflg 就很自然、很容易理解了。特殊之处在于第二个参数 nsems，它表示在由同一个信号量标识号所代表的数据结构中要设置几个信号量。也就是说，一个信号量标识号代表着一组而不只是一个信号量，由 sys_semget()建立或找到的也是一组而不只是一个信号量。看一下数据结构类型 sem_array 的定义，这一点就更清楚了（include/linux/sem.h）。

==================== include/linux/sem.h 87 97 ====================

```
87   /* One sem_array data structure for each set of semaphores in the system. */
88   struct sem_array {
89         struct kern_ipc_perm    sem_perm;/* permissions .. see ipc.h */
90         time_t                sem_otime;    /* last semop time */
91         time_t                sem_ctime;    /* last change time */
92         struct sem            *sem_base;    /* ptr to first semaphore in array */
93         struct sem_queue      *sem_pending; /* pending operations to be processed */
94         struct sem_queue      **sem_pending_last; /* last pending operation */
95         struct sem_undo            *undo;         /* undo requests on this array */
96         unsigned long         sem_nsems;    /* no. of semaphores in array */
97   };
```

显然，这一次 ipc_perm 数据结构的宿主变成 sem_array 了。同样地，整个信号量机制的总根是 ipc_ids 数据结构 sem_ids：

==================== ipc/sem.c 74 74 ====================

```
74   static struct ipc_ids sem_ids;
```

其中的指针 entries 同样指向一个 ipc_id 结构数组，而 ipc_id 结构中的指针 p 也同样指向一个 ipc_perm 数据结构，只不过它的宿主变成了 sem_array 数据结构。在这一点上，三种 SysV IPC 机制的基本格局都是一样的。可是，我们在这里注意的不是这些，而是 sem_array 结构中的指针 sem_base，它指向一个结构数组。该数组中的每一个 sem 数据结构都是一个信号量：

==================== include/linux/sem.h 81 85 ====================

```
81   /* One semaphore structure for each semaphore in the system. */
82   struct sem {
83         int  semval;        /* current value */
84         int  sempid;        /* pid of last operation */
85   };
```

这个数组的大小由参数 nsems 决定，其空间连同 sem_array 数据结构一起进行分配（123～124 行），所以紧贴在 sem_array 结构后面，而&sma[1]就是其起始地址（139 行）。

那么，为什么 sys_semget()要允许建立一组而不只是一个信号量呢？我们在讲述内核信号量时曾经

谈到临界区嵌套是很容易引起死锁的。当一项操作涉及多项共享资源时，如果先取得了其中一项，然后试图取得另一项资源不成而等待时，就可能会因为直接或间接的循环等待而形成死锁。在操作系统理论中，最典型的死锁就是由这种各自占有部分资源不放而等待其它过程释放另一部分资源而形成的所谓“哲学家与刀叉”问题。吃西餐既要用刀又要用叉，如果一个人拿到了刀不放而等待别人放弃他手上的叉，而另一个则拿到了叉不放而等待别人放弃他的刀，那就死锁了。解决的方法是：凡是要使用多项资源就一定一步（不可分割的一步）就取得所有的资源，或者在一旦得不到某项资源时就释放手中所有的相关资源。换句话说，对这些“临界资源”的取得要么是全有，要么是全无。可是，这一点对于用户空间的程序来说是难以保证的，所以要以系统调用的形式来提供这样一种机制，这就是 sys_semget()允许建立一组而不只是一个信号量的原因。这样一来，在建立了一组信号量以后，用户进程就可以通过 SEMOP 操作在一次系统调用中（因而是不可分割的）取得多项共享资源的使用权，或者就不占有任何共享资源地等待，从而防止死锁的发生。

明白了这一点，再对照 sys_msgget()的有关代码，这里 newary()的代码就很好理解了。

6.8.2 库函数 semop()——信号量操作

由于一次 SEMOP 操作可以是对一个信号量集合（而不仅仅是一个信号量）的操作，并且必须符合“要么全有，要么全无”的原则，sys_semop()的代码自然就比较复杂一些了。我们分段来看（ipc/sem.c）：

==================== ipc/sem.c 826 848 ====================

```
asmlinkage long sys_semop (int semid, struct sembuf *tsops, unsigned nsops)
{
	int error = -EINVAL;
	struct sem_array *sma;
	struct sembuf fast_sops[SEMOPM_FAST];
	struct sembuf* sops = fast_sops, *sop;
	struct sem_undo *un;
	int undos = 0, decrease = 0, alter = 0;
	struct sem_queue queue;

	if (nsops < 1 || semid < 0)
		return -EINVAL;
	if (nsops > sc_semopm)
		return -E2BIG;
	if(nsops > SEMOPM_FAST) {
		sops = kmalloc(sizeof(*sops)*nsops,GFP_KERNEL);
		if(sops==NULL)
			return -ENOMEM;
	}
	if (copy_from_user (sops, tsops, nsops * sizeof(*tsops))) {
		error=-EFAULT;
		goto out_free;
	}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的参数 tsops 是一个指针，指向用户空间中的一个 sembuf 结构数组，而 nsops 则是该数组的大小。数组中的每一项都规定了对一个信号量的操作，而对数组中所有规定的操作是“原子的”，也就是全有或全无。数据结构类型 sembuf 的定义为（include/linux/sem.h）：

==================== include/linux/sem.h 37 42 ====================

```
/* semop system calls takes an array of these. */
struct sembuf {
	unsigned short  sem_num;	/* semaphore index in array */
	short		sem_op;		/* semaphore operation */
	short		sem_flg; /* operation flags */
};
```

这里的 sem_num 为具体信号在通过 SEMGET 建立的一组信号量中的下标。而 sem_op 则为一个小小的整数，原则上这个整数会被相加到相应信号量的当前值上。如果回到我们在讨论内核信号量时所打的比喻，则当这个整数为+1 时表示退还，或多供应一张门票；而-1 则表示要取得一张门票；但是，也允许更大或更小的数值。从原理上说，这个数值的大小反映了要取得或供应同一种资源的数量。相加以后，如果具体信号量的数值变成了负数则表示不能满足要求，此时当前进程一般就会进入睡眠等待，除非要有安排（见下文）。如果 sem_op 的数值为 0，则信号量的当前值当然不会改变，而只是表示询问相应信号量的数值是否为 0，若不为 0 就等待其变成 0，除非另有安排。原则上，一个进程通过 SEMOP 操作取得的资源应该由其自己通过另一次 SEMOP 操作归还，所以如果第一次操作中对某个信号量的 sem_op 为-1，则第二次就应该是+1。此外，通过 sembuf 结构中的 sem_flg 可以设置两个标志位：一个是 IPC_NOWAIT，表示在条件不能满足时不要睡眠等待，而立即返回（出错代码为-EAGAIN）；另一个是 SEM_UNDO，表示留下遗嘱，万一当前进程欠债不还，尚未退还占有的资源就寿终正寝（exit()）的话，就由内核代为退还。

回到 sys_shmop()的代码中，从用户空间把参数复制到内核后，继续往下看（ipc/sem.c）：

==================== ipc/sem.c 849 892 ====================

```
	sma = sem_lock(semid);
	error=-EINVAL;
	if(sma==NULL)
		goto out_free;
	error = -EIDRM;
	if (sem_checkid(sma,semid))
		goto out_unlock_free;
	error = -EFBIG;
	for (sop = sops; sop < sops + nsops; sop++) {
		if (sop->sem_num >= sma->sem_nsems)
			goto out_unlock_free;
		if (sop->sem_flg & SEM_UNDO)
			undos++;
		if (sop->sem_op < 0)
			decrease = 1;
		if (sop->sem_op > 0)
			alter = 1;
	}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    alter |= decrease;

    error = -EACCES;
    if (ipcperms(&sma->sem_perm, alter ? S_IWUGO : S_IRUGO))
        goto out_unlock_free;
    if (undos) {
        /* Make sure we have an undo structure
         * for this process and this semaphore set.
         */
        un=current->semundo;
        while(un != NULL) {
            if(un->semid==semid)
                break;
            if(un->semid==-1)
                un=freeundos(sma,un);
             else
                un=un->proc_next;
        }
        if (!un) {
            error = alloc_undo(sma,&un,semid,alter);
            if(error)
                goto out_free;
        }
    } else
        un = NULL;

```

这里的 sem_lock()和 sem_checkid()与报文队列机制中的 msg_lock()和 msg_checkid()相似，读者已经熟悉了。接下来就是先对用户规定的所有信号量操作进行一番统计，看看有几项是要 SEM_UNDO 的，有几项是要改变相应信号量的当前值的。信号量也受到类似于磁盘文件一样的访问权限保护，要改变信号量的当前值就必须要具备对它的写访问权，由函数 ipcperms()加以检查。如果用户在调用 sys_semop()时至少为一个信号量操作规定了 SEM_UNDO，那就要分配一个 sem_undo 数据结构用来记录当前进程对每一组信号量的“债务”。显然，每个进程都可以有这样的“债务”，并且每个进程可以对多个信号量集合欠有这样的“债务”（要非常小心，因为这可能引起死锁），但是同一个进程对同一个信号量集合的“债务”则只要用一个数据结构就可以描述了。所以，在进程的 task_struct 中维持了一条 sem_undo 结构队列，这就是在 task_struct 结构中有个 semundo 指针的原因。

数据结构类型 sem_undo 的定义为（include/linux/sem.h）：

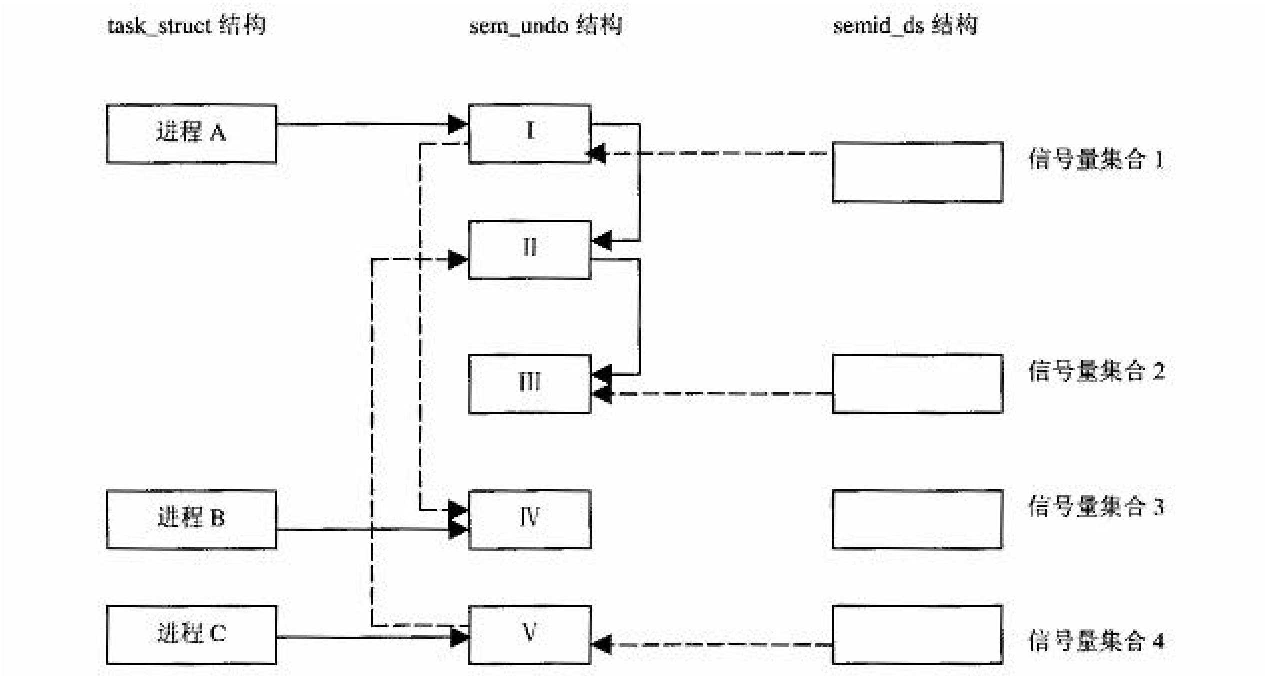
==================== include/linux/sem.h 114 122 ====================

```
/* Each task has a list of undo requests. They are executed automatically
 * when the process exits.
 */
struct sem_undo {
    struct sem_undo *  proc_next;    /* next entry on this process */
    struct sem_undo *  id_next; /* next entry on this semaphore set */
    int             semid;       /* semaphore set identifier */
    short *             semadj;       /* array of adjustments, one per semaphore */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
};
```

每一个进程要记住欠谁的债的同时，每个信号量集合也要记住都有谁欠了它的债。所以在每个信号量集合的 sem_array 中也有个指针 undo，用来维持一个 sem_undo 结构队列。而每一个 sem_undo 结构则有两个指针 proc_next 和 id_next，分别用来链入到 task_struct 结构中的队列和 sem_array 结构中的队列（见图 6.9）。函数 alloc_undo()分配一个 sem_undo 数据结构并完成两个队列的链入。



说明：每一个sem_undo结构同时在两个队列中，既属于某个进程（实线表示）又属于某个信号量集合（虚线表示），将两者联系在一起。

图 6.9　进程与信号量集合联系示意图

至此，所有的准备工作都已完成，下面就是实质性的操作了（ipc/sem.c）：

==================== ipc/sem.c 893 896 ====================

[sys_semop()]

```
	error = try_atomic_semop (sma, sops, nsops, un, current->pid, 0);
	if (error <= 0)
		goto update;

```

函数 try_atomic_semop()，正如其函数名所说那样，试图将对给定的所有信号量的操作作为一个整体来完成（ipc/sem.c）。

==================== ipc/sem.c 236 299 ====================

[sys_semop()>try_atomic_semop()]

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Determine whether a sequence of semaphore operations would succeed
 * all at once. Return 0 if yes, 1 if need to sleep, else return error code.
 */

static int try_atomic_semop (struct sem_array * sma, struct sembuf * sops,
                  int nsops, struct sem_undo *un, int pid,
                  int do_undo)
{
    int result, sem_op;
    struct sembuf *sop;
    struct sem * curr;

    for (sop = sops; sop < sops + nsops; sop++) {
        curr = sma->sem_base + sop->sem_num;
        sem_op = sop->sem_op;

        if (!sem_op && curr->semval)
            goto would_block;

        curr->sempid = (curr->sempid << 16) | pid;
        curr->semval += sem_op;
        if (sop->sem_flg & SEM_UNDO)
            un->semadj[sop->sem_num] -= sem_op;

        if (curr->semval < 0)
            goto would_block;
        if (curr->semval > SEMVMX)
            goto out_of_range;
    }

    if (do_undo)
    {
        sop--;
        result = 0;
        goto undo;
    }

    sma->sem_otime = CURRENT_TIME;
    return 0;

out_of_range:
    result = -ERANGE;
    goto undo;

would_block:
    if (sop->sem_flg & IPC_NOWAIT)
        result = -EAGAIN;
    else
        result = 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

undo:
    while (sop >= sops) {
        curr = sma->sem_base + sop->sem_num;
        curr->semval -= sop->sem_op;
        curr->sempid >>= 16;

        if (sop->sem_flg & SEM_UNDO)
            un->semadj[sop->sem_num] += sop->sem_op;
        sop--;
    }

    return result;
}
```

首先要注意到并记住，这里的参数 do_undo 为 0，这是在 sys_shmop()中的第 893 行设置好了的。所以，如果 for 循环正常结束，也就是对每个信号量的操作都没有使它的值 semval 变成负数的话，那就已经成功地取得了需要的全部资源，此时函数返回 0。除此之外，有三种情况可以使这个 for 循环中途夭折。

第一种情况是对某个信号量的操作使其数值超过了最大值 SEMVMX。在这种情况下，本次系统调用实际上不能继续下去了，所以就转到 out_of_range 处把出错代码设置成-ERANGE，然后再转到 undo 处通过一个 while 循环把前面已经完成了的操作都抵消掉，让已经在 for 循环中改变了的信号量数值都还原。不光是信号量 semval 的值要还原，表示是谁最后一次改变信号量数值的 sempid 也要还原。还有，如果 SEM_UNDO 标志为 1 的话，还要把 sem_undo 结构中记下的“账”也还原。总之一句话，是不留痕迹。

第二种情况是对某个信号量的操作使它的值变成了负数。这表示获取由这个信号量所代表的资源（的使用权）的努力受到了阻碍，一时还得不到这种资源。一般来说，这时候就要睡眠等待了，所以就转到 would_block 处。在这里，一方面根据 IPC_NOWAIT 标志的值来决定函数的返回值；另一方面同样要通过 undo 处的 while 循环将已经取得的资源全都退还，或者说将已经执行的操作都还原，也要不留痕迹。因为这里的原则是“要么全有，要么全无”。

第三种情况是对某个信号量的操作 sem_op 的值为 0，而这个信号量的当前值又是非 0，对这种情况的处理与第二种情况相同，也是转到 would_block 处。

最后，如果参数 do_undo 为非零呢？那表示只需要试一下，看看能否取得所有需要的资源，而并不是真的要改变这些信号量的数值，所以在成功以后，即 for 循环正常结束以后，就将所有的操作全部还原。

从 try_atomic_semop()返回到 sys_semop()中时，返回值有三种可能。返回值为 0 表示所有的操作都成功了，当前进程已经握有所需的全部资源，可以返回到用户空间进入“临界区”继续执行了。返回值为负值表示操作失败了，并且出了错。在这两种情况下都转到标号“ update”处，在那里进行一些善后操作，然后就返回了（见后面的代码）。第三种情况是返回值为 1，表示对某个信号量的操作失败了，需要睡眠等待。继续往下看（ipc/sem.c）：

==================== ipc/sem.c 897 964 ====================

[sys_semop()]

```
/* We need to sleep on this operation, so we put the current
 * task into the pending queue and go to sleep.
 */

queue.sma = sma;
queue.sops = sops;
queue.nsops = nsops;
queue.undo = un;
queue.pid = current->pid;
queue.alter = decrease;
queue.id = semid;
if (alter)
    append_to_queue(sma ,&queue);
else
    prepend_to_queue(sma ,&queue);
current->semsleeping = &queue;

for (;;) {
    struct sem_array* tmp;
    queue.status = -EINTR;
    queue.sleeper = current;
    current->state = TASK_INTERRUPTIBLE;
    sem_unlock(semid);

    schedule();

    tmp = sem_lock(semid);
    if(tmp==NULL) {
        if(queue.status != -EIDRM)
            BUG();
        current->semsleeping = NULL;
        error = -EIDRM;
        goto out_free;
    }
    /*
     * If queue.status == 1 we where woken up and
     * have to retry else we simply return.
     * If an interrupt occurred we have to clean up the
     * queue
     *
     */
    if (queue.status == 1)
    {
        error = try_atomic_semop (sma, sops, nsops, un,
                         current->pid,0);
        if (error <= 0)
            break;
    } else {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
945                 error = queue.status;
946                 if (queue.prev) /* got Interrupt */
947                     break;
948                 /* Everything done by update_queue */
949                 current->semsleeping = NULL;
950                 goto out_unlock_free;
951             }
952         }
953         current->semsleeping = NULL;
954         remove_from_queue(sma,&queue);
955     update:
956         if (alter)
957             update_queue (sma);
958     out_unlock_free:
959         sem_unlock(semid);
960     out_free:
961         if(sops != fast_sops)
962             kfree(sops);
963         return error;
964     }
```

与报文队列相似，睡眠时要将一个代表着当前进程的 sem_queue 数据结构链入相应 sem_array 数据结构中的 sem_pending 队列。这里 sem_queue 数据结构的定义为（include/linux/sem.h）：

==================== include/linux/sem.h 99 112 ====================

```
/* One queue for each sleeping process in the system. */
struct sem_queue {
    struct sem_queue * next;     /* next entry in the queue */
    struct sem_queue **     prev;     /* previous entry in the queue, *(q->prev) == q */
    struct task_struct*     sleeper; /* this process */
    struct sem_undo *  undo;     /* undo structure */
    int                pid;  /* process id of requesting process */
    int                status;     /* completion status of operation */
    struct sem_array * sma;  /* semaphore array for operations */
    int             id;   /* internal sem id */
    struct sembuf *         sops;     /* array of pending operations */
    int             nsops;    /* number of operations */
    int             alter;    /* operation will alter semaphore */
};
```

在将这个数据结构链入 sem_pending 队列时，还要区分本次操作是否要改变任何信号量的值，从而确定将其链入到队列的尾部或前部。处于队列前部的结构（实际上是进程），在被唤醒时享受到一些优先。此外，在进程 tasks_struct 结构中也有一个指针 semsleeping，当进程因信号量操作而进入睡眠时就指向其 sem_queue 数据结构。

进入睡眠以后，就要到被唤醒时才会从 921 行的 schedule()调用中返回。那么，由谁来唤醒呢？让我们回过头去看看前面当 try_atomic_semop()的返回值为 0 或负数时转到 update 处以后的情况。如果本次操作改变了某些信号量的值（见 956 行），那就说明这个信号量集合的状态发生了某些变化，原来因

PDF-XCHANGE Click to buy NOW! www.docu-track.com

条件不满足而只好睡眠等待的进程也许现在可以得到满足了，所以就调用 update_queue()来试试看（ipc/sem.c）：

==================== ipc/sem.c 301 330 ====================

[sys_semop()>update_queue()]

```
301  /* Go through the pending queue for the indicated semaphore
302   * looking for tasks that can be completed.
303   */
304  static void update_queue (struct sem_array * sma)
305  {
306      int error;
307      struct sem_queue * q;
308
309      for (q = sma->sem_pending; q; q = q->next) {
310
311          if (q->status == 1)
312              continue;/* this one was woken up before */
313
314          error = try_atomic_semop(sma, q->sops, q->nsops,
315                           q->undo, q->pid, q->alter);
316
317          /* Does q->sleeper still need to sleep? */
318          if (error <= 0) {
319                  /* Found one, wake it up */
320              wake_up_process(q->sleeper);
321              if (error == 0 && q->alter) {
322                  /* if q-> alter let it self try */
323                  q->status = 1;
324                  return;
325              }
326              q->status = error;
327              remove_from_queue(sma,q);
328          }
329      }
330  }
```

把 311 行的 if (q->status == 1)暂时搁一下，先看 for 循环的主体。这个循环顺着队列依次让每个正在睡眠中等待的进程试一下，看看现在能否顺利完成其信号量操作。注意，这里对 try_atomic_semop()的最后一个调用参数为 q->alter，表示如果原先的操作要改变某些信号量的值，那么现在只是试一下，而不是真的执行这些操作。试了以后的结果无非是三种：第一种是条件仍不满足，继续留在队列睡眠等待。第二种是条件仍不满足但是发生了出错，此时应将该进程唤醒，将 q->status 设置成由 try_atomic_semop()返回的出错代码，并将此数据结构从队列中摘除。既然出了错，操作已不能进行，留在队列中睡眠等待当然是毫无意义而且有害。第三种情况是某个进程的信号量操作原来因条件不满足而只好睡眠等待，但是现在条件已经能满足了，此时将该进程唤醒，并将其 q->status 设成 1，而且 for 循环就此结束。也就是说，如果队列中实际上有多个进程的条件都可以得到满足，只有排在最前面的那个进程才被唤醒。唤醒时进程的 sem_queue 数据结构仍留在队列中。另一方面，如果对同一个信

号量集合再调用一次 update_queue()，只要已被唤醒并且 q->status 已被置成 1 的进程还在队列中尚未离开，就会在 311 行的 if 语句中将其跳过。由此可见，除了在新情况下发生出错的进程不算，每次调用 update_queue()最多只唤醒一个进程，而且是排在前面的进程优先。因此，不要求改变信号量数值的进程是得到优先的，因为它们排在队列的前面。除此之外，那就是先来者优先了。

应该指出，进程本身的优先级别在这里并不起任何作用。也就是说，当有两个进程都要取得对同一组资源的使用权时，优先级别较低的进程有可能先到一步而在队列中排在较前的位置上，从而就先取得了这组资源，而优先级别较高的进程就只好“不搞特殊化”，耐心等待了。从这个意义上，严格地讲，Linux 并不是为实时系统而设计的。当然，要改变这一点也不难，这也是为什么已经有了一些“实时 Linux”版本的原因之一。

在睡眠中等待信号量操作的进程除了可以被调用 updat_queue()的进程唤醒之外，还可能因为接收到信号而被唤醒。由于进入睡眠前（见 916 行）已经把 q->status 设置成-EINTR，所以当因接收到信号而被唤醒时这个值仍然是-EINTR。还有一种情况，那就是如果一个信号量集合被取消了，此时所有正在睡眠等待的进程都会被唤醒，并且 q->status 被设置成-EIDRM。

回到 sys_semop()的代码中。从 schedule()返回以后，首先要通过 sem_lock()再次确认操作的对象仍是原先的信号量集合，并将其锁住。这个函数返回指向信号量集合的 sem_array 数据结构的指针，只有当原先的信号量集合不再存在时才返回 NULL。根据唤醒后 queue.status 的数值可以判知被唤醒的原因。这个数值为 1，表示 update_queue()发现该进程的条件满足了，但是情况也可能又有了变化（例如另一个进程已经在此之前对该信号量集合成功地进行了某些操作），所以 940 行处的 try_atomic_semop()还是有可能失败，如果失败就要同到 for 循环（见 914 行）的开始处再次入睡了。注意这里调用 try_atomic_semop()时的最后一个参数又是 0，表示这是“动真格”的。要是成功了，或者出了错，那就跳出了 for 循环，将 sem_queue 结构从队列中脱链以后就经过 update 返回了。

如果 queue.status 不为 1 呢？此时有两种可能。第一种可能是因为接收到信号而被唤醒，所以 sem_queue 结构还在队列中，queue.prev 指针必定为非 0。此时也要跳出 for 循环（947 行），也要将 sem_queue 结构从队列中脱链，并经由 update 返回。所不同的是，此时 sys_semop()的返回值必定会是-EINTR（见 916 行和 963 行）。第二种可能是因为在新的情况下发生了出错而被 update_queue()唤醒，并已从队列中脱链。此时也从 for 循环中跳出（950 行），但是直接就跳到 out_lock_free 处返回了，并且返回值就是由 update_queue()所设置的出错代码。

最后，还有个事要交待一下。我们在前面讲过，调用 sys_semop()时将 SEM_UNDO 标志设成 1 就表示“立下遗嘱”，万一进程在归还所获取的资源之前就 exit()的话，就委托内核代为归还。实际上，既然占有资源的进程已经 exit()，则这些资源事实上已经不再被占用了。问题是相应信号量的数值没有得到调整，就好像仓库里明明有东西但账本上却说没有了。而委托内核做的事，就是平时每次都通过 sem_undo 数据结构记下账，然后当进程 exit()时，根据该进程的 sem_undo 数据结构队列来调整有关信号量的数值。这是怎样实现的呢？读者可以回到第 4 章 do_exit()的代码中看一下，在那里要调用一个函数 sem_exit()。这个函数所做的事情是：如果 exit()的进程中某个信号量在集合的队列中等待，就将其脱链。然后扫描该进程的 semundo 队列，根据每个 sem_undo 数据结构中的记载，依次对相应信号量集合中的相应信号量数值作出调整。最后调用 update_queue()唤醒可能正在等待的进程。函数的代码也在 sem.c 中，我们把它留给读者自己阅读。

6.8.3 库函数 semctl()——信号量的控制与管理

与 msgctl()大同小异。函数 sys_semctl()的代码虽然不短，逻辑却很简单。我们把它留给读者了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第7章 基于 socket 的进程间通信

7.1 系统调用 socket()

相对于传统的 Unix IPC，“插口”，即 socket（有些资料中也称“套接字 ”)，是更为一般的进程间通信机制。它既适用于同一台计算机上的进程间通信，也适用于网络环境的进程间通信，并且是当今所有网络操作系统必不可少的基础功能。本章侧重从同一计算机上的进程间通信的角度介绍 socket 在 Linux 内核中的实现。至于它在网络环境中的推广，则因分量太重、篇幅太大，只好留作另一本书的内容。

在 Unix 的发展史中，AT&T 的贝尔实验室与加州大学伯克利分校的伯克利软件发布中心（BSD）可以说是两大主力。当 AT&T 致力于改进传统的 Unix 进程间通信功能，从而形成了一整套 SysV IPC 机制的同时，BSD 也在设法对其加以改进。与此同时，BSD 又最早将计算机网络的通信规程，特别是当时正在成形的 TCP/IP 规程，实现到 Unix 的内核中去。所以，对于当时的 BSD 来说，很自然地会把二者结合在一起考虑，把同一计算机（或同网络“节点”）上的进程间通信纳入更广的、网络范围的进程间通信范畴，从而设计出一种更为一般化的进程间通信机制。这种努力的结果就是 socket 机制，这一机制其实是命名管道在计算机网络环境下的实现和推广。

如果比较一下 AT&T 和 BSD 各自在这方面的努力，就可以看出，AT&T 是有系统地、全面地对传统的 Unix IPC 加以改进。例如，针对在某些应用中（传统 Unix IPC 的）效率不够高的缺点，设计和开发了共享内存机制；针对缺乏进程间同步手段的问题，又设计和开发了用户空间的信号量机制。另外，管道要占用打开文件号，便引入了“键值”的概念，从而避开了使用打开文件号。但是，AT&T 的眼光始终只盯着单一计算机中的进程间通信。而 BSD 则正好相反，它对传统 Unix 进程间通信的改进并不是有系统的和全面的，可是它的眼光却跳出了单机的范围。虽然 socket 机制在单机范围内与 AT&T 的报文传递机制在概念上极为相似，但是却更为一般化，从而为后来网络技术的蓬勃发展做好了技术准备。可以说，SysV IPC 和 socket 是互相补充而不是各搞一套。

至于 Linux，则很自然地兼收并蓄，把二者都继承下来了。

顾名思义，一个 socket 就好像一个通信线的插口。只要通信的双方都有插口，并且，两个插口之间有通信线路相连接，就可以互相通信了。从概念上说，socket 与管道其实并无多大区别，如果把两个插口之间的“连线”比喻成“管道”，那么插口就相当于管道两端的“水龙头”，而且二者都表现为已打开文件。不同的是管道两端只能在同一台计算机上，而通过虚拟的“通信线路”相连接的两个插口却可以分别存在于计算机网络中的不同节点上。当然，尽管二者在概念上相似，在具体的实现和使用上却有着很大的不同。而且，管道所传递的是无结构的字节流，而通过 socket 传递的则是有结构的报文。

一个插口在逻辑上有三个特征，或者说三个要素，那就是网域、类型以及规程。

首先是网域，它表明一个插口是用于哪一种网络，或者说哪一族网络规程的。由于各种网络对节

点地址的命名方法不同，所以又称为“地址族”（address family）或“规程族”（protocol family）。例如，常数 AF_INET 表示互联网（英特网）插口，所以各节点都使用 IP 地址；而 AF_IPX 则为 Novell 的 IPX 网插口，AF_X25 为 X.25 网插口，等等。其中有个特例，那就是什么网也不是，只是在一台计算机上用于进程间通讯，BSD 为这种特例定义的域名为 AF_UNIX。后来，在 POSIX 标准里又定义了一个 AF_LOCAL，以示对别的操作系统也一视同仁。

其次为“类型”，它表明在网络中通信所遵循的模式。网络通信有两种主要模式，一种称为“有连接”或“面向连接”（connection oriented）的通信；另一种则称为“无连接”（connectionless）通信。“有连接”模式常常又称为“虚电路”（virtual circuit）模式。在这种模式中，通信的双方要先通过一定的步骤在互相之间建立起一种虚拟的连接，或者说虚拟的线路，然后再通过虚拟的连接线路进行通信。在通信的过程中，所有报文传递都保持着原来的次序，报文在网络传输的过程中受到的不均匀延迟会在接收端得到补偿。所以所有报文之间都是有关联的，每个报文都不是孤立的。更重要的是，在这种模式中所有报文的传递都是“可靠”的，由网络中物理通信线路引入的差错会由通信规程中的应答和重发机制加以克服。同时，在这种模式中还提供了“流量控制”的手段。从用户的角度来看，“有连接”类型的插口对报文的传递作出了承诺。如果一个进程通过系统调用在一个“有连接”插口上发送一个报文，那么，只要进程从这次系统调用正常返回，就说明该报文已经被递交到了接收方的插口（但接收方进程未必已经读取这个报文）。“无连接”模式就不同了。“无连接”模式常常又称为“数据包”（datagram）模式，也称“面向报文”的通信模式（message oriented）。在“无连接”模式中并不需要事先在双方之间建立起“虚电路”，而直接就可以发送或接收报文，但是每个报文都是孤立的，其正确性也没有保证，甚至可能丢失。如果两个报文穿越网络时走过了不同的路径，或者甚至相同的路径，但是受到了不一致的延迟，那么它们在接收端的次序就可能与发送端的次序不同。所以，由“无连接”模式的插口所提供的报文传递是不可靠的，它对用户作出的承诺只是“尽力传递”，但却是没有保证的。此外，在“无连接”模式中也没有“流量控制”手段。如果一个进程通过系统调用在一个“无连接”插口上发送一个报文，那么当进程从这次系统调用正常返回时只是说明该插口将会例行公事地将报文传递到接收方，但并不表示这报文已经到达了接收方的插口。从另一个角度，还可以说，“有连接”插口的报文传递是同步的，而“无连接”插口的报文传递则是异步的。

最后是“规程”，它表明具体的网络规程。一般来说，网域和类型结合在一起大体上就确定了适用的规程。例如，要是网域为 AF_INET，而类型为“无连接”，则规程基本上就是 UDP 了。但是，在有些情况下还可能会有别的选择，此时就由它来进一步明确具体的规程。

其实，插口的这三个特征是互相联系的，归根结底就是反映了一个插口所运行的（网络）规程。由于在每个插口的后面都隐藏着网络规程，对插口的比较详尽的讨论就势必要涉及到计算机网络这个课题。要在本书中谈论计算机网络，并进而分析 Linux 内核中有关网络规程的代码是不现实的，因为那本身就已经足够构成另一本书的材料了。所以，我们在本书中将只讨论 Unix 域的插口，也就是用于同一计算机中进程间通信的插口，隐藏在这种插口后而的规程是“没有规程”。

Unix 域的插口同样也分两种模式，但是二者都提供可靠的报文传递。在网络环境下，不可靠性是由网络的基础设施（如物理线路）引起的，两种不同模式实际上反映了对待物理介质的不可靠性的不同态度和策略。“有连接”模式采取了“大包大揽”的态度，企图在不可靠物理介质的基础上构筑起一层可靠的报文传递机制。而“无连接”模式则采取了“矛盾上交”的态度，说“既然物理介质不可靠，那我也没有办法，只能有什么给你什么”，让用户自己去设法克服或避免由此而引起的问题。可是，Unix

PDF-XCHANGE Click to buy NOW! www.docu-track.com

域的插口既然只提供同一台计算机上的进程间通信，而根本就不涉及网络设施，其物理介质实际上就是内存，那就根本不存在由物理介质引入的不可靠性。不过，尽管 Unix 域插口的两种模式都提供可靠的报文传递，二者还是有区别的，读完本节以后就会清楚这些区别。

像对 Sys V IPC 操作一样，Linux 内核为所有与 socket 有关的操作提供一个统一的系统调用入口，但是在用户程序界面上则通过 C 语言程序库 c.1ib 提供诸多库函数，看起来就好像都是独立的系统调用一样。内核中为 socket 设置的总入口为 sys_socketcall()，其代码在 net/socket.c 中：

==================== net/socket.c 1512 1596 ====================

```
/*
 *   System call vectors.
 *
 *   Argument checking cleaned up. Saved 20% in size.
 *  This function doesn't need to set the kernel lock because
 *  it is set by the callees.
 */

asmlinkage long sys_socketcall(int call, unsigned long *args)
{
     unsigned long a[6];
     unsigned long a0,a1;
     int err;

     if(call<1||call>SYS_RECVMSG)
          return -EINVAL;

     /* copy_from_user should be SMP safe. */
     if (copy_from_user(a, args, nargs[call]))
          return -EFAULT;

     a0=a[0];
     a1=a[1];

     switch(call)
     {
          case SYS_SOCKET:
               err = sys_socket(a0,a1,a[2]);
               break;
          case SYS_BIND:
               err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
               break;
          case SYS_CONNECT:
               err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
               break;
          case SYS_LISTEN:
               err = sys_listen(a0,a1);
               break;
          case SYS_ACCEPT:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
            break;
        case SYS_GETSOCKNAME:
            err = sys_getsockname(a0,(struct sockaddr *)a1, (int *)a[2]);
            break;
        case SYS_GETPEERNAME:
            err = sys_getpeername(a0, (struct sockaddr *)a1, (int *)a[2]);
            break;
        case SYS_SOCKETPAIR:
            err = sys_socketpair(a0,a1, a[2], (int *)a[3]);
            break;
        case SYS_SEND:
            err = sys_send(a0, (void *)a1, a[2], a[3]);
            break;
        case SYS_SENDTO:
            err = sys_sendto(a0,(void *)a1, a[2], a[3],
                      (struct sockaddr *)a[4], a[5]);
            break;
        case SYS_RECV:
            err = sys_recv(a0, (void *)a1, a[2], a[3]);
            break;
        case SYS_RECVFROM:
            err = sys_recvfrom(a0, (void *)a1, a[2], a[3],
                        (struct sockaddr *)a[4], (int *)a[5]);
            break;
        case SYS_SHUTDOWN:
            err = sys_shutdown(a0,a1);
            break;
        case SYS_SETSOCKOPT:
            err = sys_setsockopt(a0, a1, a[2], (char *)a[3], a[4]);
            break;
        case SYS_GETSOCKOPT:
            err = sys_getsockopt(a0, a1, a[2], (char *)a[3], (int *)a[4]);
            break;
        case SYS_SENDMSG:
            err = sys_sendmsg(a0, (struct msghdr *) a1, a[2]);
            break;
        case SYS_RECVMSG:
            err = sys_recvmsg(a0, (struct msghdr *) a1, a[2]);
            break;
        default:
            err = -EINVAL;
            break;
    }
    return err;
}
```

函数的第一个参数 call 即为具体的操作码，而参数 args 为指向一个数组的指针。根据具体操作码的不同，需要从用户空间复制参数的数量也不同。为了根据操作码确定需要从用户空间复制的字节数，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

代码 net/socket.c 中定义了一个数组 nargs[]：

==================== net/socket.c 1505 1510 ====================
```
/* Argument list sizes for sys_socketcall */
#define AL(x) ((x) * sizeof(unsigned long))
static unsigned char nargs[18]={AL(0),AL(3),AL(3),AL(3),AL(2),AL(3),
                                AL(3),AL(3),AL(4),AL(4),AL(4),AL(6),
                                AL(6),AL(2),AL(5),AL(5),AL(3),AL(3)};
#undef AL
```

至于操作码，则是在 include/linux/net.h 定义的：

==================== include/linux/net.h 30 46 ====================
```
#define SYS_SOCKET 1          /* sys_socket(2)          */
#define SYS_BIND   2          /* sys_bind(2)            */
#define SYS_CONNECT     3          /* sys_connect(2)         */
#define SYS_LISTEN 4          /* sys_listen(2)          */
#define SYS_ACCEPT 5          /* sys_accept(2)          */
#define SYS_GETSOCKNAME 6          /* sys_getsockname(2)         */
#define SYS_GETPEERNAME 7          /* sys_getpeername(2)         */
#define SYS_SOCKETPAIR  8          /* sys_socketpair(2)          */
#define SYS_SEND   9          /* sys_send(2)            */
#define SYS_RECV   10         /* sys_recv(2)            */
#define SYS_SENDTO 11         /* sys_sendto(2)          */
#define SYS_RECVFROM    12         /* sys_recvfrom(2)        */
#define SYS_SHUTDOWN    13         /* sys_shutdown(2)        */
#define SYS_SETSOCKOPT  14         /* sys_setsockopt(2)          */
#define SYS_GETSOCKOPT  15         /* sys_getsockopt(2)          */
#define SYS_SENDMSG     16         /* sys_sendmsg(2)         */
#define SYS_RECVMSG     17         /* sys_recvmsg(2)         */
```

注释中括号里的“2”表示所述的函数为系统调用。

我们先把这些操作码（以及相应的函数）分一下类，然后说明它们的用途，最后再来看它们是怎样实现的。

这些操作码（函数）大体上可分为五类。

7.1.1 插口的创建与撤除

属于这一类的操作码有：

- SYS_SOCKET：创建一个插口，其用户程序界面（由 libc 提供，下同）为：

  int socket(int domain, int type, int protocol);

  这里的三个参数即为前面所述插口的三个要素。不过，通常第三个参数 protocol 为 0，因为一般来说前两个参数确定以后，具体的规程也就定了，所以用 0 表示默认由系统根据前两个参数确定的规程，只有在比较特殊的应用中才需要指定具体的规程。插口创建成功以后返回一个正整数，实际上是一个打开文件号。但这个打开文件号是与一个代表插口的数据结构相联

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

系的，并不是与磁盘上的某个文件相联系。

- SYS_BIND：将代表着一个插口的打开文件号与某一个域中的可寻址实体或“插口地址”相结合。例如，在互联网中的可寻址实体为网中以 IP 地址区分的节点，而在同一节点上又有若干不同的“传输层”收发口（逻辑意义上），所以此时的结合就是与 IP 地址加收发口逻辑编号的结合。在 Unix 域中，此种可寻址实体是一个文件名，所以就成了与文件名的结合。用户程序界面为：

  int bind(int soekfd, struct sockaddr *my_addr, socklen_t addrlen);

  这里的 sockfd 即为 socket()所返回的打开文件号。在 bind 到某个地址或文件名之前，虽然插口已经建立但却无从寻访。如果说 socket()好像是安装了一个电话机的话，那么 bind()就好像为它指定了一个电话号码。
- SYS_SOCKETPAIR：创建一对已经互相连接的无名插口，概念上类似于 pipe()。用户程序界面为：

  int socketpair(int domain, int type, int protocal, int sv[2]);

  前面三个参数与 socket()相同，数组 sv[]则用来返回创建后的一对打开文件号。
- SYS_SHUTDOWN：部分或完全关闭一个插口，用户程序界面为：

  int shutdown(int s, int how);

  参数 s 为插口的打开文件号，参数 how 为 0 表示不再允许接收，1 表示不再允许发送，2 则表示接收和发送都不再允许。由于插口体现为特殊的打开文件，所以也可以用 close()来关闭。

7.1.2 插口间连接的建立

“有连接”模式通信的双方是不对称的，可以说天生就是 client/server 的关系。而连接的建立则必须经过一定的步骤：

- SYS_LISTEN：作为 server 的一方首先要通过 listen()向内核“挂号”。只有挂了号的插口才会被内核视为 server 一侧的插口并允许它接受来自 client 一侧的连接请求。用户程序界面为：

  int listen(int s, int backlog);

  参数 s 即为插口的打开文件号。当连接请求到来，而一时得不到接受时，就暂时进入一个队列，在那里等待 server 方的接受。这里的参数 backlog 就规定了这个队列的最大长度。
- SYS_CONNECT：在“有连接”模式中，client 一方要通过 connect()向已经挂了号的 server 方插口请求连接。用户程序界面为：

  int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

  参数 sockfd 为 client 一侧插口的打开文件号，serv_addr 则指向一个表示 server 方插口地址的数据结构。当然，这个地址必须已经由 server 方通过 bind()指定给 server 方的插口。由于有些网域中的地址可能不是固定长度的，所以还要有个参数 addrlen 来指明其长度。

  对 connect()的调用一般要到 server 方接受了连接或者出错时才返回。
- SYS_ACCEPT：server 方通过 accept()接受或等待接受到来的（或已经在队列中的）第一个连接请求。用户程序界面为：

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

参数 s 为一个已经挂了号的 server 方插口（打开文件号），参数 addr 与 addrlen 则用来返回连接请求的来源，也就是 client 方插口的地址。对 accept()调用要到有连接请求到来而建立起连接，或者发生了出错时才返回。

特别要说明的是，这个函数返回一个新的打开文件号。这个新的打开文件号就代表着已经建立起连接的 server 方插口。而原来的插口，则保持原样不变，又可以用来接受新的（其它的）连接请求。这个插口就好像是下蛋的鸡，每接受一个连接请求就生下一个"蛋"，那就是已经建立起连接的插口。住典型的应用中，server 进程在通过 accept()接受了一个连接请求，从而与一个 client 进程建立起一个连接以后，就会通过 fork()创建出一个子进程。然后，子进程将作为"种籽"的 server 方插口关闭，而使用新的插口与 client 进程通信并为之提供服务。而父进程则把新的插口关闭，并且再一次调用 accept()，通过"种籽"插口来接受新的连接请求。这就是典型的 client/server 运行模式。

7.1.3 "有连接"模式的报文发送与接收

"有连接"模式的插口一定要在 client 和 server 双方建立起连接以后才能用于通信。由于插口在用户界面上表现为已打开文件，并且"有连接"模式的通信既是可靠的又保持原有的次序，所以可以把传递的信息看作有序的字节流，从而可以用普通的 read()和 write()系统调用，像读 / 写普通文件一样地来接收和发送信息。除此之外，也可以用专门为插口而设的三对库函数之一来接收和发送，即 recv()/send()、recvfrom()/sendto()以及 recvmsg()/sendmsg()。不过，这些库函数主要用于"无连接"模式，所以我们将它们与"无连接"模式的发送与接收放在一起介绍。这里还要指出，无论是"有连接 "还是"无连接"模式，插口都是双向的，并且就发送和接收而言双方是对称的。

7.1.4 "无连接"模式的报文发送与接收

顾名思义，"无连接"模式的插口不需要事先建立连接，插口一经创建马上就可以发送或接收报文。另一方面，由于"无连接"模式的通信既不是可靠的，也不一定保持原有的次序，所以不宜用 read()和 write()像对待一个有序字节流那样使用"无连接"模式的插口，而要用专门设置的三组库函数来进行。与文件操作 read( )、write()相比，这三组函数的特点是它们都保留报文的边界，有关这一点读者在后面看了源代码以后就清楚了。

- SYS_SENDTO：如前所述，"无连接"模式的插口一经建立（并且 bind 到一个插口地址）以后就可以进行通信，而无需先建立连接。当然，此时对所发送的每个报文都要提供对方的地址（在"无连接"模式中，每个报文都是独立的），所以，sendto()的用户程序界面为：

  int sendto(int s, const void *msg, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);

  显然，这个函数是专为"无连接"模式的报文发送而设置的，参数 to 指向对方的地址即 sockaddr

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

数据结构，而 tolen 则为地址的长度。虽然这个函数的界面是专为“无连接”模式设计的，但是也可以用这个函数在“有连接”模式的插口上发送报文，不过此时应将参数 to 设置成 NULL，而将 tolen 设置成 0。

在内核中，操作 SYS_SENDTO 是由 sys_sendto()实现的。

- SYS_SEND：从用户程序界面来看，send()似乎主要是为“有连接”模式设计的：

    int send(int s, const void *msg, size_t len, int flags);

  与 sendto()的界面相比，这里没有提供对方的地址。在“有连接”模式中，连接已经事先建立好，当然不需要每次都提供对方地址了。但是，即使在“无连接”模式中，当准备接连向同一目标发送很多个报文时，每次都要提供对方的地址。这样做既麻烦又降低了效率（每次都要从用户空间把地址复制到内核中）。是不是可以简化一下呢？例如，是否可以先“预设”一个双方地址，随后就采用 send()来发送，而不必每次都重复地提供相同的地址。事实正是这样，对于“无连接”模式的插口，可以用 connect()先设置一个对方地址，然后再用 send()发送报文，而实际上每次都使用预先设置好的对方地址。但是要注意，在“无连接”模式中使用 connect()与在“有连接”模式中使用 connect()有本质的区别。在“无连接”模式中，connect()的作用只是让内核为“本地”插口记下预设的对方地址，而并不涉及与对方之间控制报文的往返。以后则在发送的每个报文头部附上这个地址，以指明报文的目的地。至于在“有连接”模式中的 connect()，则实际向对方发送一个请求连接的控制报文（指在网络环境下），并等待对方的响应。连接建立了以后，随同每个报文发送的可以只是一个连接号，而不一定要包括对方地址。所以，虽然在形式上两种模式都可以使用 connect()，并且有时把经过 connect()预设好对方地址的“无连接”插口也说成是处于“已连接状态”，但实质上是完全不同的。读者在资料中碰到此类词句时要注意根据上下文加以区分。

  在内核中，SYS_SEND 是由 sys_send()实现的，而 sys_send()又是由 sys_sendto()实现的，代码见 net/socket.c：

==================== net/socket.c 1207 1210 ====================

[sys_socketcall()>sys_send()]

```
1207 asmlinkage long sys_send(int fd, void * buff, size_t len, unsigned flags)
1208 {
1209      return sys_sendto(fd, buff, len, flags, NULL, 0);
1210 }
```

- SYS_SENDMSG：库函数 sendmsg()是前两个函数的推广与加强，这是三个函数中最复杂的，其用户程序界面为：

    int sendmsg(int s, const struct msghdr *msg, int flags);

  参数 msg 指向一个 msghdr 数据结构，定义于 include/linux/net.h：

==================== include/linux/socket.h 33 41 ====================

```
33   #define SYS_LISTEN 4          /* sys_listen(2)        */
34   #define SYS_ACCEPT 5          /* sys_accept(2)        */
35   #define SYS_GETSOCKNAME 6        /* sys_getsockname(2)        */
36   #define SYS_GETPEERNAME 7        /* sys_getpeername(2)        */
37   #define SYS_SOCKETPAIR  8        /* sys_socketpair(2)         */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define SYS_SEND    9          /* sys_send(2)           */
#define SYS_RECV    10         /* sys_recv(2)           */
#define SYS_SENDTO 11          /* sys_sendto(2)      */
#define SYS_RECVFROM    12         /* sys_recvfrom(2)      */
```

每个 msghdr 数据结构都代表着一个报文。在 msghdr 结构中，msg_name 为对方的插口地址（或者也可以称作插口名）。而 msg_iov 则指向一个结构数组，该数组中的每一个元素都是一块数据，这样一个报文的内容就可以分散在若干个互不连续的缓冲区中，而在逻辑上却连在一起，在网络环境下这是很有好处的。还有，msg_control 和 msg_controllen 的作用是传递控制信息，在 Unix 域中用来在进程间传递访问权限，还可以用来传递“打开文件描述体”。这些以后再介绍。

由于每个报文都有个对方地址，这显然适合于“无连接”模式。但是，像 sendto()一样，它也可以用于“有连接”模式中，不过要将 msg_name 设成 NULL，msg_namelen 设成 0。

- SYS_RECV、SYS_RECVFROM、SYS_RECVMSG：这三个操作都是用来从某个插口接收报文的，并且分别与前列用来发送报文的操作相对应，所以我们把它们合并在一起叙述。完成这些操作的库函数为：

    int recv(int s, void *bur, size_t len, int flags);

    int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);

    int recvmsg(int s, struct msghdr *msg, int flags);

    与发送报文的库函数类似，recv()通常用于“有连接”模式的插口，或者虽然是“无连接”模式的插口，但却已经用 connect()预设了对方地址。函数 recvfrom()的接收是全方位的，参数 from 并不是用来指定接收来自哪一个远方插口的报文，而是用来在接收到一个报文时指明报文的来源。所以 from 所指向的数据结构只是一个用于返回报文来源的缓冲区，而 fromlen 所指则为该缓冲区的大小。在内核中，SYS_RECV 由 sys_recv()实现，SYS_RECVFROM 则由 sys_recvfrom()实现。但是，就像 sys_send()与 sys_sendto()之间的关系一样，sys_recv()也是通过 sys_recvfrom()实现的，代码见 net/socket.c：

==================== net/socket.c 1258 1261 ====================

[sys_socketcall()>sys_recv()]

```
asmlinkage long sys_recv(int fd, void * ubuf, size_t size, unsigned flags)
{
    return sys_recvfrom(fd, ubuf, size, flags, NULL, NULL);
}
```

也就是说，如果把 recvfrom()的参数 from 和 fromlen 设置成 NULL，就跟 recv()一样了。函数 recvmsg()则与 sendmsg()相对应，也具有将一个报文的内容分散在若干个缓冲区内的功能，并且具有接收控制信息的功能。

7.1.5 控制以及辅助性的操作

- SYS_GETSOCKNAME：库函数 getsockname()用来获取为一个插口 s 指定（bind）地址或名字：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

int getsockname(int s, struct sockaddr *name, socklen_t *namelen);

- SYS_GETPEERNAME："有连接"模式的插口，不管是 client 方还是 server 方，一旦建立起连接以后就有了一个"对方"。库函数 getpeername()用来获取插口 s 的对方插口的地址（或名字）：

  int getpeername(int s, struct sockaddr *name, socklen_t *namelen);

- SYS_SETSOCKOPT、SYS_GETSOCKOPT：库函数 setsockopt()和 getsockopt()用来设置和读取一个插口 s 所实行规程中的一些可选项，由于可选项都是在网络环境下运用的，而本书只讲述 Unix 域的插口，所以我们在这里并不关心这两个函数。

有了上面这些预备知识以后，我们就可以从下一节开始介绍 socket 通信机制在 Unix 域中的实现了。为了往后阅读的方便，此处先给出利用插口实现进程间通信的流程示意图(图 7.1)。

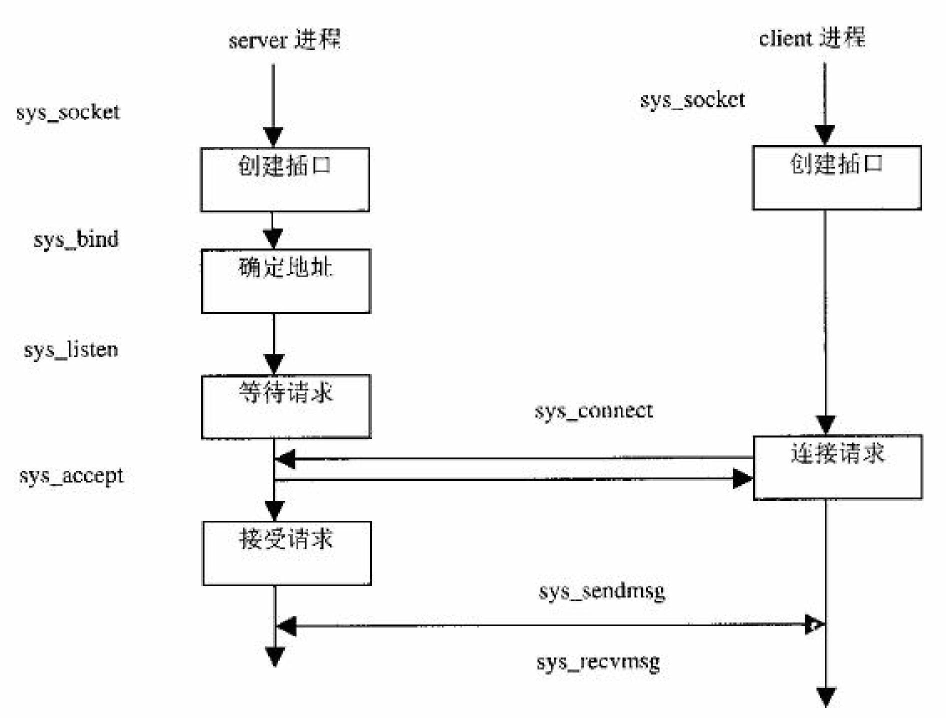


图 7.1　插口通信流程示意图

7.2　函数 sys_socket()——创建插口

操作 SYS_SOCKET 是由 sys_socket()实现的，其代码在 net/socket.c 中：

==================== net/socket.c 889 909 ====================
[sys_socketcall()>sys_socket()]

```
889  asmlinkage long sys_socket(int family, int type, int protocol)
890  {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int retval;
    struct socket *sock;

    retval = sock_create(family, type, protocol, &sock);
    if (retval < 0)
        goto out;

    retval = sock_map_fd(sock);
    if (retval < 0)
        goto out_release;

out:
    /* It may be already another descriptor 8) Not kernel problem. */
    return retval;

out_release:
    sock_release(sock);
    return retval;
}
```

前面说过，插口对于用户程序而言就是特殊的已打开文件。内核中为插口定义了一种特殊的文件类型，形成一种特殊的文件系统 sockfs，定义于 net/socket.c 中：

==================== net/socket.c 301 303 ====================

```
static struct vfsmount *sock_mnt;
static DECLARE_FSTYPE(sock_fs_type, "sockfs", sockfs_read_super,
    FS_NOMOUNT|FS_SINGLE);
```

在系统初始化时，要通过 kern_mount()安装这个文件系统。安装时有个作为连接件的 vfsmount 数据结构，这个结构的地址就保存在一个全局的指针 sock_mnt 中。所谓创建一个插口，就是在 sockfs 文件系统中创建一个特殊文件，或者说一个节点，并建立起为实现插口功能所需的一整套数据结构。所以首先是建立一个 socket 数据结构，然后将其“映射”到一个已打开文件中。函数 sock_create()的代码在同一文件（net/socket.c）中。这段代码由于比较长，我们分段来看：

==================== net/socket.c 814 859 ====================

[sys_socketcall()>sys_socket()>sock_create()]

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
    int i;
    struct socket *sock;

    /*
     *    Check protocol is in range
     */
    if(family<0 || family>=NPROTO)
        return -EAFNOSUPPORT;

    /* Compatibility.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

        This uglymoron is moved from INET layer to here to avoid
        deadlock in module load.
       */
      if (family == PF_INET && type == SOCK_PACKET) {
           static int warned;
           if (!warned) {
                warned = 1;
                printk(KERN_INFO "%s uses obsolete (PF_INET,SOCK_PACKET)\n", current->comm);
           }
           family = PF_PACKET;
      }

#if defined(CONFIG_KMOD) && defined(CONFIG_NET)
      /* Attempt to load a protocol module if the find failed.
       *
       * 12/09/1996 Marcin: But! this makes REALLY only sense, if the user
       * requested real, full-featured networking support upon configuration.
       * Otherwise module support will break!
       */
      if (net_families[family]==NULL)
      {
           char module_name[30];
           sprintf(module_name,"net-pf-%d",family);
           request_module(module_name);
      }
#endif

      net_family_read_lock();
      if (net_families[family] == NULL) {
           i = -EAFNOSUPPORT;
           goto out;
      }

```

这一段代码的开始部分是检查和处理参数的范围。由于我们在这里只关心 Unix 域，也就是当 family 为 AF_UNIX 时的情景，所以这段代码对我们不起什么作用。接下来是一段条件编译，如果系统配置了可动态安装内核模块的功能，并且网络驱动程序是动态安装的，就要检查一下由参数 family 所指定网域的驱动程序是否已经安装，尚未安装的话就要调用 request_module()把它安装起来。至于 Unix 域插口的驱动程序，那是内核所固有的，并非动态安装的模块。此外，就像每种文件系统都有个 file_operations 数据结构一样，每种网域，包括 Unix 域，也都有个 net_proto_family 数据结构。在系统初始化或安装模块时，要将指向相应网域的这个数据结构的指针填入一个数组 net_families[]中，否则就说明系统尚不支持给定的网域。Unix 域的 net_proto_family 数据结构为 unix_family_ops，定义于 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1844 1847 ====================

```
struct net_proto_family unix_family_ops = {
```

```
    PF_UNIX,
    unix_create
};
```

就是说，结构中只有一个代表着 Unix 域的常数 PF_UNIX 和一个函数指针 unix_create，而 PF_UNIX 决定了指向这个数据结构的指针在 net_families[]中的位置。

回到 sys_socket()中继续往下看（net/socket.c）：

==================== net/socket.c 860 887 ====================

[sys_socketcall()>sys_socket()>sock_create()]

```
/*
 *  Allocate the socket and allow the family to set things up. if
 *  the protocol is 0, the family is instructed to select an appropriate
 *  default.
 */

    if (!(sock = sock_alloc()))
    {
        printk(KERN_WARNING "socket: no more sockets\n");
        i = -ENFILE;       /* Not exactly a match, but its the
                             closest posix thing */
        goto out;
    }

    sock->type  = type;

    if ((i = net_families[family]->create(sock, protocol)) < 0)
    {
        sock_release(sock);
        goto out;
    }

    *res = sock;

out:
    net_family_read_unlock();
    return i;
}
```

插口是由 socket 数据结构代表的，这种数据结构定义于 include/linux/net.h 中：

==================== include/linux/net.h 65 79 ====================

```
struct socket
{
    socket_state       state;

    unsigned long      flags;
    struct proto_ops   *ops;
    struct inode       *inode;
```

```
	struct fasync_struct	*fasync_list; /* Asynchronous wake up list	*/
	struct file		*file;		/* File back pointer for gc */
	struct sock		*sk;
	wait_queue_head_t	wait;

	short			type;
	unsigned char		passcred;
};
```

结构中各个成分的用途随着代码的阅读会变得清楚起来，这里暂且只要知道有这些成分就可以了。不过我们建议读者在搞清了这些成分的用途以后再回过头来自己加上注释。

函数 sock_alloc()分配一个 socket 数据结构并进行一些初始化，其代码在 net/socket.c 中：

==================== net/socket.c 427 463 ====================

[sys_socketcall()>sys_socket()>sock_create()>sock_alloc()]

```
/**
 *	sock_alloc	-	allocate a socket
 *
 *	Allocate a new inode and socket object. The two are bound together
 *	and initialised. The socket is then returned. If we are out of inodes
 *	NULL is returned.
 */

struct socket *sock_alloc(void)
{
	struct inode * inode;
	struct socket * sock;

	inode = get_empty_inode();
	if (!inode)
		return NULL;

	inode->i_sb = sock_mnt->mnt_sb;
	sock = socki_lookup(inode);

	inode->i_mode = S_IFSOCK|S_IRWXUGO;
	inode->i_sock = 1;
	inode->i_uid = current->fsuid;
	inode->i_gid = current->fsgid;

	sock->inode = inode;
	init_waitqueue_head(&sock->wait);
	sock->fasync_list = NULL;
	sock->state = SS_UNCONNECTED;
	sock->flags = 0;
	sock->ops = NULL;
	sock->sk = NULL;
	sock->file = NULL;

```

```
461       sockets_in_use[smp_processor_id()].counter++;
462       return sock;
463  }
```

可见，取得一个 inodc 结构是取得一个 socket 结构的必要条件。不仅如此，socket 结构其实只是 inode 结构中的一部分！读者在“文件系统”一章中看到过有关 inode 数据结构的说明，在 inode 结构中有一个关键性的成分 u。这是一个 union，要按具体的文件类型和格式而解释成不同的数据结构。目前 Linux 支持 20 多种不同的文件系统，因此对这个 union 有 20 多种不同的解释，而 socket 结构正是其中之一。代码中 445 行的 socki_lookup()只是将 inode 结构中的这个 union 解释为 socket 结构而已，见 net/socket.c 中的代码：

==================== net/socket.c 377 380 ====================

[sys_socketcall()>sys_socket()>sock_alloc()>socki_lookup()]

```
377  extern __inline__ struct socket *socki_lookup(struct inode *inode)
378  {
379       return &inode->u.socket_i;
380  }
```

同时，在 inode 结构中还要将 i_mode 里的 S_IFSOCK 标志位设成 1，并将 i_sock 也设成 1，以示这个 inode 结构所代表的并不是磁盘文件，而是一个插口。

分配了一个 socket 结构，并且设置好插口的类型以后，就通过由 unix_family_ops 提供的函数指针 create 调用 Unix 域的插口创建程序 unix_create()，其代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 498 523 ====================

[sys_socketcall()>sys_socket()>sock_create()>unix_create()]

```
498  static int unix_create(struct socket *sock, int protocol)
499  {
500       if (protocol && protocol != PF_UNIX)
501            return -EPROTONOSUPPORT;
502
503       sock->state = SS_UNCONNECTED;
504
505       switch (sock->type) {
506       case SOCK_STREAM:
507            sock->ops = &unix_stream_ops;
508            break;
509            /*
510             *   Believe it or not BSD has AF_UNIX, SOCK_RAW though
511             *   nothing uses it.
512             */
513       case SOCK_RAW:
514            sock->type=SOCK_DGRAM;
515       case SOCK_DGRAM:
516            sock->ops = &unix_dgram_ops;
517            break;
518       default:
519            return -ESOCKTNOSUPPORT;
```

```
    }

    return unix_create1(sock) ? 0 : -ENOMEM;
}
```

参数 protocol 是从用户空间一直传下来的，若为 0 则默认为 Unix 域。插口的初始状态设置成 SS_UNCONNECTED，“有连接”模式，即类型为 SOCK_STREAM 的插口必须在建立了连接以后才能使用。插口的类型决定了通信的模式，对于给定的网域这通常也决定了采用的规程。对插口的各种操作因规程的不同而异，所以各种规程都有其自己的一套插口操作，通过一个 proto_ops 数据结构提供有关的函数指针。根据插口类型的不同，这里把 socket 结构中的指针分别设置成 unix_stream_ops 或 unix_dgram_ops。这两个数据结构均定义于 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1804 1842 ====================

```
struct proto_ops unix_stream_ops = {
    family:     PF_UNIX,

    release:  unix_release,
    bind:       unix_bind,
    connect:  unix_stream_connect,
    socketpair:   unix_socketpair,
    accept:     unix_accept,
    getname:  unix_getname,
    poll:       unix_poll,
    ioctl:      unix_ioctl,
    listen:     unix_listen,
    shutdown: unix_shutdown,
    setsockopt:   sock_no_setsockopt,
    getsockopt:   sock_no_getsockopt,
    sendmsg:  unix_stream_sendmsg,
    recvmsg:  unix_stream_recvmsg,
    mmap:       sock_no_mmap,
};

struct proto_ops unix_dgram_ops = {
    family:     PF_UNIX,

    release:  unix_release,
    bind:       unix_bind,
    connect:  unix_dgram_connect,
    socketpair:   unix_socketpair,
    accept:     sock_no_accept,
    getname:  unix_getname,
    poll:       datagram_poll,
    ioctl:      unix_ioctl,
    listen:     sock_no_listen,
    shutdown: unix_shutdown,
    setsockopt:   sock_no_setsockopt,
    getsockopt:   sock_no_getsockopt,
```

```
    sendmsg: unix_dgram_sendmsg,
    recvmsg: unix_dgram_recvmsg,
    mmap:        sock_no_mmap,
};
```

对比一下上述这两个数据结构，就可以看出在无连接模式插口中对应于 accept 的函数为 sock_no_accept()，这个函数只是返回一个出错代码-EOPNOTSUPP，表示不支持所要求的操作。同样地，也不支持 listen。同时，两种模式的 Unix 域插口都不支持 setsockopt 和 getsockopt，也不支持 mmap。不过，如前所述，对于用户程序而言，插口就是已打开文件，所以还可能通过常规的文件操作界面支持一些比较通用的操作，有些操作如 unix_poll()、unix_ioctl()等，并不是通过 sys_socketcall()来调用的，而要通过普通的文件操作界面来调用。

此外，一般网域的插口类型除 SOCK_STREAM 和 SOCK_DGRAM 外还有一种 SOCK_RAW，也是无连接模式的，在 Unix 域中则等同于 SOCK_DGRAM，所以 514 行把插口类型改成 SOCK_DGRAM，注意在 514 行下面没有 break 语句。

最后，unix_create()调用 unix_create1()，进一步完成创建插口的任务（net/unix/af_unix.c）：

==================== net/unix/af_unix.c 464 496 ====================

[sys_socketcall()>sys_socket()>sock_create()>unix_create()>unix_create1()]

```
static struct sock * unix_create1(struct socket *sock)
{
    struct sock *sk;

    if (atomic_read(&unix_nr_socks) >= 2*files_stat.max_files)
        return NULL;

    MOD_INC_USE_COUNT;
    sk = sk_alloc(PF_UNIX, GFP_KERNEL, 1);
    if (!sk) {
        MOD_DEC_USE_COUNT;
        return NULL;
    }

    atomic_inc(&unix_nr_socks);

    sock_init_data(sock,sk);

    sk->write_space        =    unix_write_space;

    sk->max_ack_backlog = sysctl_unix_max_dgram_qlen;
    sk->destruct = unix_sock_destructor;
    sk->protinfo.af_unix.dentry=NULL;
    sk->protinfo.af_unix.mnt=NULL;
    sk->protinfo.af_unix.lock = RW_LOCK_UNLOCKED;
    atomic_set(&sk->protinfo.af_unix.inflight, 0);
    init_MUTEX(&sk->protinfo.af_unix.readsem);/* single task reading lock */
    init_waitqueue_head(&sk->protinfo.af_unix.peer_wait);
    sk->protinfo.af_unix.list=NULL;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    unix_insert_socket(&unix_sockets_unbound, sk);

    return sk;
}
```

每个插口都有个 socket 数据结构，同时又有个 sock 数据结构，后者可以说是对前者的一种扩充，两者间是一对一的对应关系。在 socket 结构中有个指针 sk 指向其对应的 sock 结构，而在 sock 结构中则有个指针 socket 指向其对应的 socket 结构，所以二者可以说是同一个东西的两个侧面。其中 sock 结构是一种相当大的数据结构，其定义有 180 多行，我们不在这里列出了，读者可以自己在 include/linux/net/sock.h 中找到它的定义。在这里，我们只是随着代码的进展对有关的成分作一些介绍。另一方面，sock 结构是内核中常常要动态分配使用的，所以内核中为此专设了一个队列，通过 slab 机制来管理这种数据结构的缓冲区。分配了 sock 结构以后就是它的初始化，sock_init_data()的代码也在 net/socket.c 中：

==================== net/core/sock.c 1117 1157 ====================

[sys_socketcall()>sys_socket()>sock_create()>unix_create()>unix_create1()>sock_init_data()]

```

asmlinkage long sys_getsockname(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
{
    struct socket *sock;
    char address[MAX_SOCK_ADDR];
    int len, err;

    sock = sockfd_lookup(fd, &err);
    if (!sock)
        goto out;
    err = sock->ops->getname(sock, (struct sockaddr *)address, &len, 0);
    if (err)
        goto out_put;
    err = move_addr_to_user(address, len, usockaddr, usockaddr_len);

out_put:
    sockfd_put(sock);
out:
    return err;
}

/*
 *  Get the remote address ('name') of a socket object. Move the obtained
 *  name to user space.
 */

asmlinkage long sys_getpeername(int fd, struct sockaddr *usockaddr, int *usockaddr_len)
{
    struct socket *sock;
    char address[MAX_SOCK_ADDR];
    int len, err;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    if ((sock = sockfd_lookup(fd, &err))!=NULL)
    {
        err = sock->ops->getname(sock, (struct sockaddr *)address, &len, 1);
        if (!err)
            err=move_addr_to_user(address,len, usockaddr, usockaddr_len);
        sockfd_put(sock);
    }
    return err;
}
```

在 sock 结构中有几个双向链队列，其中最重要的就是 receive_queue 和 write_queue，而 error_queue 则仅在网络环境下才会用到。这几个队列并不采用通用的队列头结构 list_head，而专门定义了一种 sk_buff_head 数据结构，定义于 include/linux/skbuff.h 中：

==================== include/linux/skbuff.h 51 58 ====================

```
struct sk_buff_head {
    /* These two members must be first. */
    struct sk_buff    * next;
    struct sk_buff    * prev;

    __u32         qlen;
    spinlock_t    lock;
};
```

排在队列中的都是 sk_buff 数据结构。在网络环境下，我们通常所说的“报文”是 ISO 的 7 层模型中第 4 层，即“传输层”的概念；而具体在网络中发送和接收的则是第 3 层，即“网络层”的数据单位，称为 packet（“报文分组”，或“包”）。根据长度，一个报文可以就是一个包，也可以在发送端被分解成若干个包，而在接收端予以组装复原。报文与包之间的这种区分主要是为克服网络中物理介质的不可靠性，以及从性能方面考虑而来的。在 Unix 域的条件下，由于并不涉及网络介质，所以一个报文就是一个包。每一个包都要占用一个 sk_buff 数据结构，所以 receive_queue 队列中的每个 sk_buff 数据结构就载运着一个到达的包，而 write_queue 队列中则为待发送的包。这也是一种比较复杂的数据结构，我们将结合报文的发送和接收加以介绍。

此外，在 sock 结构中还有一个特殊的 sk_buff 结构队列，那是专为网络环境而设置的，我们在这里并不关心。在网络环境下，常常要为包的发送设置一些定时器。例如，在“有连接”模式中如果发送了一个包以后在一定时间里得不到对方的确认就要重发。所以在 sock 结构中还有一个定时器队列 timer。所有这些队列的头部都要加以初始化(见 1125～1130 行)。

函数 sock_init_data()中其余的代码比较直截了当，我们就不多解释了。只说明一下，sock 结构中的 rcvbuf 和 sndbuf 分别为接收和发送缓冲区的大小，默认值均为 64K 字节。还有，sock 结构中的 state_change、data_ready 等都是函数指针，分别设置成指向 sock_def_wakeup()、sock_def_readable()等函数。

回到 unix_create1()的代码中，继续完成 sock 结构的初始化。注意代码中的 sock 是一个 socket 结构指针，sk 才是 sock 结构指针。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的 sk->write_space 也是一个函数指针，设置成指向函数 unix_write_space()。在 sock 数据结构中还有一个非常重要的成分 protinfo，这是个 union，要根据具体网域而赋予不同的解释。对 Unix 域来说，这个 union 被当作一个 unix_opt 数据结构，名为 af_unix，这个数据结构是在 include/net/sock.h 中定义的：

==================== include/net/sock.h 106 118 ====================

```
/* The AF_UNIX specific socket options */
struct unix_opt {
	struct unix_address	*addr;
	struct dentry *		dentry;
	struct vfsmount *	mnt;
	struct semaphore	readsem;
	struct sock *		other;
	struct sock **		list;
	struct sock *		gc_tree;
	atomic_t		inflight;
	rwlock_t		lock;
	wait_queue_head_t	peer_wait;
};
```

这里的 addr 指向一个 unix_address 结构。将一个插口 bind 到某一个地址时，这个地址就保存在这里，其类型定义在 include/net/af_unix.h 中：

==================== include/net/af_unix.h 20 26 ====================

```
struct unix_address
{
	atomic_t	refcnt;
	int		len;
	unsigned	hash;
	struct sockaddr_un name[0];
};
```

具体的地址在 sockaddr_un 结构中，这个数据结构的定义在 include/linux/un.h 中：

==================== include/linux/un.h 4 9 ====================

```
#define UNIX_PATH_MAX	108

struct sockaddr_un {
	sa_family_t sun_family;	/* AF_UNIX */
	char sun_path[UNIX_PATH_MAX];	/* pathname */
};
```

可见，Unix 域中的插口地址即为一个文件（节点）的路径名。注意在 unix_address 结构中的 name[] 数组大小为 0，所以 unix_address 结构的大小并不包括 sockaddr_un 结构的空间，在分配空间时要额外加上。那么，为什么要这样呢？这是因为插口的地址可长可短（最长为 108 个字节），如果固定按最大长度分配空间就太浪费了。

除 addr 以外，af_unix 结构中的 readsem 是个内核信号量，代码中的 init_ MUTEX()将这个内核信

PDF-XCHANGE Click to buy NOW! www.docu-track.com

号量的值初始化成 1，用来保证某些操作对于资源的独占性。

对 sock 结构初始化的最后一步是 unix_insert_socket()，这是个 inline 函数，其定义为：

==================== net/unix/af_unix.c 241 246 ====================

[sys_socketcall()>sys_socket()>sock_create()>unix_create()>unix_create1()>unix_insert_socket()]

```
static __inline__ void unix_insert_socket(unix_socket **list, unix_socket *sk)
{
    write_lock(&unix_table_lock);
    __unix_insert_socket(list, sk);
    write_unlock(&unix_table_lock);
}
```

函数__unix_insert_socket()则将一个 sock 数据结构插入到一个 unix_socket 结构队列中，并将该 sock 结构的使用计数设置成 1。内核中设置了一个杂凑表 unix_socket_table[]，其大小为 UNIX_HASH_SIZE+1，即 257。定义见 net/unix/af_unix.c：

==================== net/unix/af_unix.c 118 118 ====================

```
unix_socket *unix_socket_table[UNIX_HASH_SIZE+1];
```

这个数组中的每一项都是一个 unix_socket 结构的单链队列。在 include/net/afunix.h 中有对 unix_socket 的定义：

==================== include/net/af_unix.h 6 6 ====================

```
typedef struct sock unix_socket;
```

所以 unix_socket 结构就是 sock 结构。每个 sock 结构根据其插口地址的杂凑值挂入这个杂凑表中的某个队列。这样，当接收到一个无连接模式的报文时，或接收到一个连接请求时，就可以根据其目标地址迅速地找到其目标插口。但是，在插口创建之初尚无插口地址，所以暂时把它链入到杂凑表中的最后一个队列，即 unix_socket_table[UNIX_HASH_SIZE]。由于地址的杂凑值都在[0, UNIX_HASH_SIZE-1]范围内，所以不会引起混淆。代码中的 unix_sockets_unbound 在 net/unix/af_unix.c 中定义为：

==================== net/unix/af_unix.c 122 122 ====================

```
#define unix_sockets_unbound     (unix_socket_table[UNIX_HASH_SIZE])
```

当然，它在这个队列中只是暂时栖身而已。到用户程序调用 bind()的时候，就会根据地址的杂凑值把插口的 sock 结构转移到杂凑表中的另一个队列中去。

前面讲过，socket 结构与 sock 结构实际上是同一事物的两个侧面。不妨说，socket 结构是面向进程和系统调用界面的侧面，而 sock 结构则是面向底层驱动程序的侧面。可是，为什么不干脆把两个数据结构合并成一个呢？如前所述，socket 结构是 inode 结构中的一部分，即把 inode 结构内部的一个 union 用作 socket 结构。由于插口操作的特殊性，这个数据结构中需要有大量的结构成分。可是，如果把这些结构成分全都放在 socket 结构中，则 inode 结构中的这个 union 就会变得很大，从而 inode 结构也会变得很大，而对于其它文件系统这个 union 是不需要那么庞大的。所以，那样会使 inode 结构变得过分庞大而造成浪费，系统中使用 inode 结构的数量当然要远远超过使用 socket 结构的数量。解决的办法就是把插口所需的这些结构成分拆成两部分，把与文件系统关系比较密切的那一部分放在 socket 结构中，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

另一部分，即与通信关系比较密切的那一部分，则单独成为一个数据结构，那就是 sock 结构。由于这两部分数据在逻辑上本来就是一体的，所以要通过指针互相指向对方，形成一对一的关系。

完成了 socket 结构和 sock 结构的分配以及初始化，sock_create()就完成了任务，我们回到 sys_socket()中继续往下看。下一步是 sock_map_fd()，代码见 net/socket.c：

==================== net/socket.c 312 375 ====================

[sys_socketcall()>sys_socket()>sock_map_fd()]

```
/*
 *   Obtains the first available file descriptor and sets it up for use.
 *
 *   This functions creates file structure and maps it to fd space
 *   of current process. On success it returns file descriptor
 *   and file struct implicitly stored in sock->file.
 *   Note that another thread may close file descriptor before we return
 *   from this function. We use the fact that now we do not refer
 *   to socket after mapping. If one day we will need it, this
 *   function will inincrement ref. count on file by 1.
 *
 *   In any case returned fd MAY BE not valid!
 *   This race condition is inavoidable
 *   with shared fd spaces, we cannot solve is inside kernel,
 *   but we take care of internal coherence yet.
 */

static int sock_map_fd(struct socket *sock)
{
	int fd;
	struct qstr this;
	char name[32];

	/*
	 *   Find a file descriptor suitable for return to the user.
	 */

	fd = get_unused_fd();
	if (fd >= 0) {
		struct file *file = get_empty_filp();

		if (!file) {
			put_unused_fd(fd);
			fd = -ENFILE;
			goto out;
		}

		sprintf(name, "[%lu]", sock->inode->i_ino);
		this.name = name;
		this.len = strlen(name);
		this.hash = sock->inode->i_ino;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
353
354             file->f_dentry = d_alloc(sock_mnt->mnt_sb->s_root, &this);
355             if (!file->f_dentry) {
356                 put_filp(file);
357                 put_unused_fd(fd);
358                 fd = -ENOMEM;
359                 goto out;
360             }
361             file->f_dentry->d_op = &sockfs_dentry_operations;
362             d_add(file->f_dentry, sock->inode);
363             file->f_vfsmnt = mntget(sock_mnt);
364
365             sock->file = file;
366             file->f_op = sock->inode->i_fop = &socket_file_ops;
367             file->f_mode = 3;
368             file->f_flags = O_RDWR;
369             file->f_pos = 0;
370             fd_install(fd, file);
371         }
372
373     out:
374         return fd;
375     }
```

前面讲过，从进程的角度来看，一个插口就是一个特殊的已打开文件，这是在设计“插口”这种机制时就设计好了的。现在 socket 结构（以及 sock 结构）已经分配好并进行了初始化，接着要做的就是将 socket 结构，实际上是其宿主 inode 结构，与文件系统的一套机制挂上钩了。首先要分配一个空闲的打开文件号以及 file 结构；还要在文件系统的目录树中分配一个 dentry 数据结构，使其指向已经分配的 inode 数据结构；并且使 file 结构中的指针 f_dentry 指向该 dentry 结构。最后，还会根据所建立的 dentry 结构和 inode 结构在磁盘上建立起相应的目录项和索引节点，不过那是后话。从代码中还可看出，代表着插口的节点名就是其索引节点号（349 行）。

代码中 354 行的 d_alloc()为插口分配一个目录项，并将其挂在特殊文件系统 sockfs 的根目录下。然后，362 行的 d_add()将插口的 dentry 结构与 inodc 结构互相挂上钩。这里把 deniry 结构中的指针 d_op 设置成指向 sockfs_dentry_operations，这个数据结构通过函数指针提供了与文件路径有关的操作，定义于 net/socket.c 中：

==================== net/socket.c 308 310 ====================

```
308     static struct dentry_operations sockfs_dentry_operations = {
309         d_delete: sockfs_delete_dentry,
310     };
```

可见，对于插口，除 vfs 层上的操作以外惟一与此有关的操作就是删除目录项。

最后，将 file 结构中的 f_op 指针和 inode 结构中的 i_fop 指针都设置成指向用于插口的文件操作跳转结构 socket_fs_ops，并根据分配得到的打开文件号 fd 将该 file 结构的地址填入本进程的打开文件结构数组中相应的位置上。数据结构 socket_file_ops 的定义为（net/socket.c）：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== net/socket.c 114 126 ====================

```
static struct file_operations socket_file_ops = {
	llseek:		sock_lseek,
	read:		sock_read,
	write:		sock_write,
	poll:		sock_poll,
	ioctl:		sock_ioctl,
	mmap:		sock_mmap,
	open:		sock_no_open, /* special open code to disallow open via /proc */
	release:	sock_close,
	fasync:		sock_fasync,
	readv:		sock_readv,
	writev:		sock_writev
};
```

这样，当通过 sys_socketcall()对一个插口进行某种操作时（如 recv()等），就会通过前述 socket 结构中的 unix_stream_ops 或 unix_dgram_ops 结构跳转到具体的驱动程序中；而通过普通的文件操作界面（如 read()等）对插口操作时，则通过这个 socket_fs_ops 结构决定跳转。

完成了 sock_mapfd()以后，整个创建插口的过程就完成了。总结以上所述，我们可以粗略地画出内核中与插口有关的主要数据结构之间的联系图（图 7.2）。

图中有两个入口。一个是进程的 task_struct 数据结构，根据打开文件号可以找到相应的 inode 结构；另一个是从杂凑表 unix_socket_table 开始找到相应的 sock 结构。当接收到一个 packet 时，在 packet 的头部必然直接或间接地包含着关于目标插口地址的信息。根据这个地址，加以杂凑运算后便可以确定 unix_socke_table 表中的一个队列，然后扫描该队列加以比对，就可以找到属于该目标地址的 sock 数据结构，而找到了 sock 结构也就找到了 socket 结构以及 inode 结构。

最后还要说明，除 SYS_SOCKET 以后，还有个 SYS_SOCKETPAIR 操作也用来创建插口。SYS_SOCKETPAIR 所建立的是一对（而不是一个）互相已经连接好的插口，概念上与“管道”相似。使用上也很相似（例如都要与 fork()结合使用）。事实上，在有些 Unix 版本中甚至用“插口对”来实现“管道”。由于“插口对”只能在同一台计算机上使用，就没有必要再根据报文的目标地址通过杂凑表找到目标插口的 sock 结构，而可以把这一步“短路”过去了。为了这个目的，在 sock 数据结构中设置了一个指针 pair。在创建一个“插口对”时就让两个 sock 数据结构都通过这个指针互相指向对方。SYS_SOCKETPAIR 操作是由 sys_socketpair()实现的，有兴趣的读者可以自行阅读其代码。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

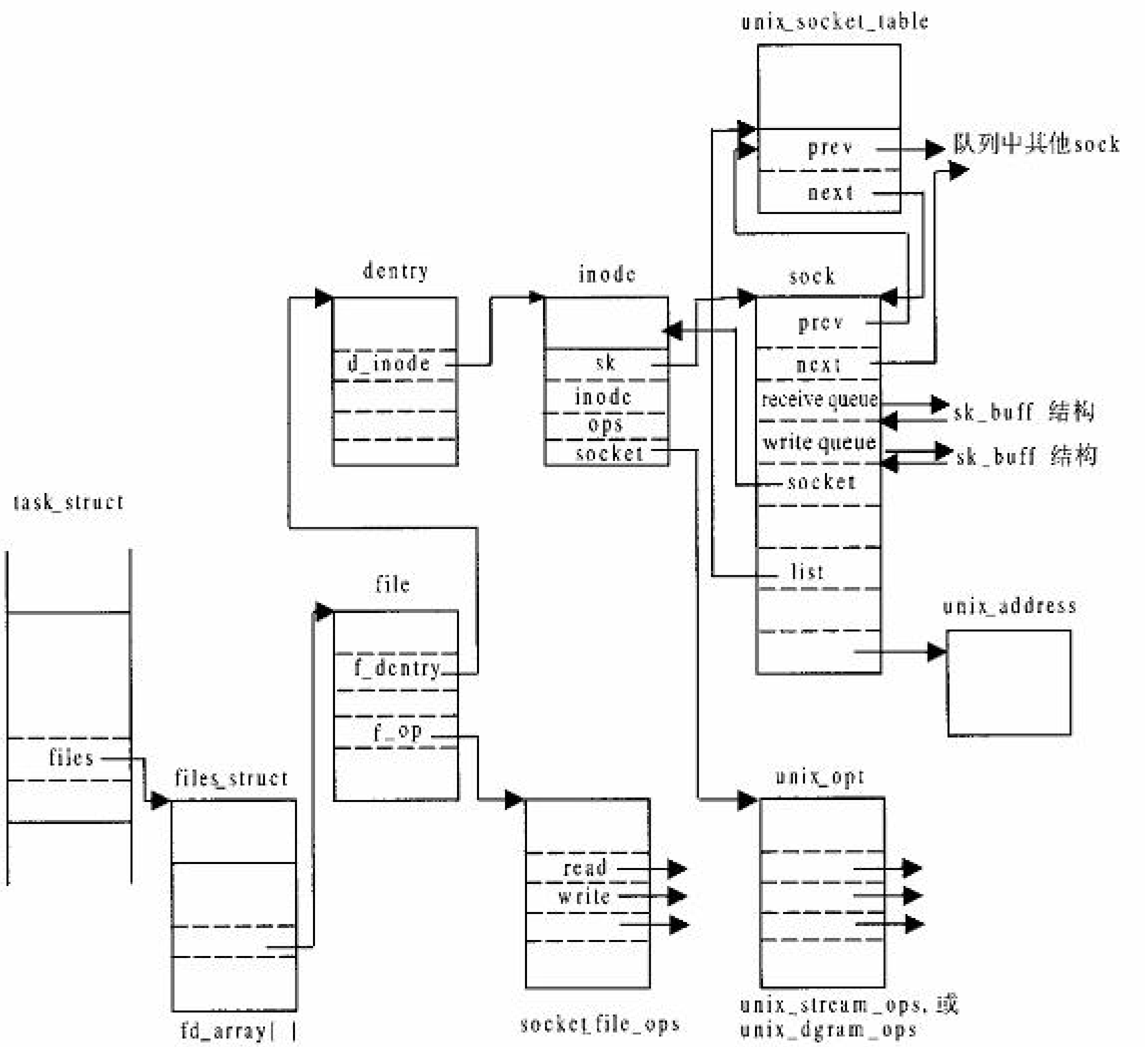


图 7.2　与 socket 相关的主要数据结构之间的联系图

7.3　函数 sys_bind()——指定插口地址

操作 SYS_BIND 为已创建的插口指定一个地址，是由 sys_bind()实现的（net/socket.c）:

==================== net/socket.c 977 998 ====================

[sys_socketcall()>sys_bind()]

```
/*
 *	Bind a name to a socket. Nothing much to do here since it's
 *	the protocol's responsibility to handle the local address.
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *	We move the socket address to kernel space before we call
 *	the protocol layer (having also checked the address is ok).
 */

asmlinkage long sys_bind(int fd, struct sockaddr *umyaddr, int addrlen)
{
	struct socket *sock;
	char address[MAX_SOCK_ADDR];
	int err;

	if((sock = sockfd_lookup(fd,&err))!=NULL)
	{
		if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0)
			err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
		sockfd_put(sock);
	}
	return err;
}
```

首先要根据插口的打开文件号找到它的 socket 数据结构，sockfd_lookup()的代码很简单，读者可以结合前面的图 7.2 自行阅读（net/socket.c）。

==================== net/socket.c 382 420 ====================

[sys_socketcall()>sys_bind()>sockfd_lookup()]

```
/**
 *	sockfd_lookup -	Go from a file number to its socket slot
 *	@fd: file handle
 *	@err: pointer to an error code return
 *
 *	The file handle passed in is locked and the socket it is bound
 *	too is returned. If an error occurs the err pointer is overwritten
 *	with a negative errno code and NULL is returned. The function checks
 *	for both invalid handles and passing a handle which is not a socket.
 *
 *	On a success the socket object pointer is returned.
 */

struct socket *sockfd_lookup(int fd, int *err)
{
	struct file *file;
	struct inode *inode;
	struct socket *sock;

	if (!(file = fget(fd)))
	{
		*err = -EBADF;
		return NULL;
	}

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    inode = file->f_dentry->d_inode;
    if (!inode->i_sock || !(sock = socki_lookup(inode)))
    {
        *err = -ENOTSOCK;
        fput(file);
        return NULL;
    }

    if (sock->file != file) {
        printk(KERN_ERR "socki_lookup: socket file changed!\n");
        sock->file = file;
    }
    return sock;
}
```

找到了插口的 socket 结构以后，就可以根据该结构中的指针 ops 找到相应的驱动程序跳转结构 unix_stream_ops 或 unix_dgram_ops。两个跳转结构中的指针 bind 都指向 unix_bind()，所以 Unix 域的 bind 操作是由 unix_bind()完成的。

不过，先要把参数 umyaddr 所指的数据结构从用户空间复制过来，这个数据结构含有为插口指定的地址。前面在 sys_socketcall()复制过来的只是指向这个数据结构的指针，而不是数据结构本身。这里（988 行）为插口的地址准备下了一个缓冲区 address[]，其大小为 MAX_SOCK_ADDR，定义为 128。具体的复制是由 move_addr_to_kernel()完成的，复制的长度则取决于参数 addrlen。

参数 umyaddr 是个 sockaddr 结构指针，我们还没有看过这个数据结构的定义，现在需要看一下了，这是在文件 include/linux/socket.h 中给出的：

==================== include/linux/socket.h 11 20 ====================

```
typedef unsigned short sa_family_t;

/*
 *  1003.1g requires sa_family_t and that sa_data is char.
 */

struct sockaddr {
    sa_family_t   sa_family;    /* address family, AF_xxx   */
    char      sa_data[14];  /* 14 bytes of protocol address */
};
```

虽然 sockaddr 数据结构有确定的大小，其中的 sa_data[]是一个 14 字节的数组，但是插口地址的实际长度却并不是固定的，所以由另一个参数 addrlen 说明其包括 sa_family 在内的长度。读者也许感到奇怪：sa_data[]中不是一个字符串吗？只要用 strlen()取字符串长度就行了，何必这么麻烦？其实不然，就 Unix 域来说，这通常是个字符串，可是在其它网域中就未必了，例如在 internet 网域中就可以是 4 个字节的 IP 地址加上传输层的端口号，其中有些字节完全可能是 0（后面读者会看到，即使在 Unix 域中也还有所谓“抽象地址”，是以“\0”开头的）。另一方面，Unix 域的插口地址实际上是个文件路径名，14 个字节显然是不够的。所以，用于插口地址的数据结构大小因网域而异，可是结构中的第一个成分总是 sa_family。这么一考虑，读者就可以明白当初设计者（从 BSD Unix 时代开始）的用意了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在极端的情况下，地址的长度 addrlen 可以只包含了数据类型 sa_family 的长度，也就是一个短整数的长度，而 sa_data[]中则连一个字节也没有。设计者赋予这样的地址一个特殊的含义，就是让系统给自动分配（生成）一个地址。对于 Unix 域插口地址，实际使用的是 sockaddr_un 结构，与 sockaddr 结构的区别在于它的字符数组大小为 UNIX_PATH_MAX，实际上是 108。注意 address[]的大小为 MAX_SOCK_ADDR，即 128，已经考虑到了对所有网域插口地址的需要。由于在 address[]中有足够大的空间，将 sockaddr_un 结构复制过来不会造成问题。常数 MAX_SOCK_ADDR 的定义和 move_addr_to_kernel()的代码都在 net/socket.c 中：

==================== net/socket.c 194 223 ====================

```
#define MAX_SOCK_ADDR  128       /* 108 for Unix domain -
                                 16 for IP, 16 for IPX,
                                 24 for IPv6,
                                 about 80 for AX.25
                                 must be at least one bigger than
                                 the AF_UNIX size (see net/unix/af_unix.c
                                 :unix_mkname()).
                               */

/**
 *   move_addr_to_kernel    -    copy a socket address into kernel space
 *   @uaddr: Address in user space
 *   @kaddr: Address in kernel space
 *   @ulen: Length in user space
 *
 *   The address is copied into kernel space. If the provided address is
 *   too long an error code of -EINVAL is returned. If the copy gives
 *   invalid addresses -EFAULT is returned. On a success 0 is returned.
 */

int move_addr_to_kernel(void *uaddr, int ulen, void *kaddr)
{
    if(ulen<0||ulen>MAX_SOCK_ADDR)
        return -EINVAL;
    if(ulen==0)
        return 0;
    if(copy_from_user(kaddr,uaddr,ulen))
        return -EFAULT;
    return 0;
}
```

其主体是 copy_from_user()，读者对此已不陌生了。

如前所述，Unix 域的 bind 操作是由 unix_bind()完成的，其代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 636 768 ====================

[sys_socketcall()>sys_bind()>unix_bind()]

```
static int unix_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct sock *sk = sock->sk;
    struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
    struct dentry * dentry = NULL;
    struct nameidata nd;
    int err;
    unsigned hash;
    struct unix_address *addr;
    unix_socket **list;

    err = -EINVAL;
    if (sunaddr->sun_family != AF_UNIX)
        goto out;

    if (addr_len==sizeof(short)) {
        err = unix_autobind(sock);
        goto out;
    }

    err = unix_mkname(sunaddr, addr_len, &hash);
    if (err < 0)
        goto out;
    addr_len = err;

    down(&sk->protinfo.af_unix.readsem);

    err = -EINVAL;
    if (sk->protinfo.af_unix.addr)
        goto out_up;

    err = -ENOMEM;
    addr = kmalloc(sizeof(*addr)+addr_len, GFP_KERNEL);
    if (!addr)
        goto out_up;

    memcpy(addr->name, sunaddr, addr_len);
    addr->len = addr_len;
    addr->hash = hash^sk->type;
    atomic_set(&addr->refcnt, 1);

    if (sunaddr->sun_path[0]) {
        err = 0;
        /*
         * Get the parent directory, calculate the hash for last
         * component.
         */
        if (path_init(sunaddr->sun_path, LOOKUP_PARENT, &nd))
            err = path_walk(sunaddr->sun_path, &nd);
        if (err)
            goto out_mknod_parent;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
        /*
         * Yucky last component or no last component at all?
         * (foo/., foo/.., /////)
         */
        err = -EEXIST;
        if (nd.last_type != LAST_NORM)
            goto out_mknod;
        /*
         * Lock the directory.
         */
        down(&nd.dentry->d_inode->i_sem);
        /*
         * Do the final lookup.
         */
        dentry = lookup_hash(&nd.last, nd.dentry);
        err = PTR_ERR(dentry);
        if (IS_ERR(dentry))
            goto out_mknod_unlock;
        err = -ENOENT;
        /*
         * Special case - lookup gave negative, but... we had foo/bar/
         * From the vfs_mknod() POV we just have a negative dentry -
         * all is fine. Let's be bastards - you had / on the end, you've
         * been asking for (non-existent) directory. -ENOENT for you.
         */
        if (nd.last.name[nd.last.len] && !dentry->d_inode)
            goto out_mknod_dput;
        /*
         * All right, let's create it.
         */
        err = vfs_mknod(nd.dentry->d_inode, dentry,
            S_IFSOCK|sock->inode->i_mode, 0);
        if (err)
            goto out_mknod_dput;
        up(&nd.dentry->d_inode->i_sem);
        dput(nd.dentry);
        nd.dentry = dentry;

        addr->hash = UNIX_HASH_SIZE;
    }

    write_lock(&unix_table_lock);

    if (!sunaddr->sun_path[0]) {
        err = -EADDRINUSE;
        if (__unix_find_socket_byname(sunaddr, addr_len,
                        sk->type, hash)) {
            unix_release_addr(addr);
            goto out_unlock;
```

```
736             }
737 
738             list = &unix_socket_table[addr->hash];
739         } else {
740             list = &unix_socket_table[dentry->d_inode->i_ino & (UNIX_HASH_SIZE-1)];
741             sk->protinfo.af_unix.dentry = nd.dentry;
742             sk->protinfo.af_unix.mnt = nd.mnt;
743         }
744 
745         err = 0;
746         __unix_remove_socket(sk);
747         sk->protinfo.af_unix.addr = addr;
748         __unix_insert_socket(list, sk);
749 
750     out_unlock:
751         write_unlock(&unix_table_lock);
752     out_up:
753         up(&sk->protinfo.af_unix.readsem);
754     out:
755         return err;
756 
757     out_mknod_dput:
758         dput(dentry);
759     out_mknod_unlock:
760         up(&nd.dentry->d_inode->i_sem);
761     out_mknod:
762         path_release(&nd);
763     out_mknod_parent:
764         if (err==-EEXIST)
765             err=-EADDRINUSE;
766         unix_release_addr(addr);
767         goto out_up;
768     }
```

对于 Unix 域的插口地址，代码中先把它的 sockaddr 结构当成一个 sockaddr_un 结构（638 行），后者的字符数组大小为 UNIX_PATH_MAX，即 108。此时实际的插口地址已经在 sys_bind()中的局部量 address[]中，其大小为 128，所以不会造成问题。

如前所述，如果参数 addrlen 指示插口地址的大小为 2，即仅为数据类型 sa_family_t 的大小，则表示要求由系统给自动分配（生成）一个地址。这就是 652 行中对 unix_autobind()的调用。这个函数的代码也在 af_unix.c 中，建议读者在按正常路线读完 unix_bind()的代码以后自己再看一下 unix_autobind()。

虽然 move_addr_to_kernel()对插口地址的长度作了检查，对其内容却并未触及，所以要进一步通过 unix_mkname()检查和处理。顺便提一句，变量名"sunaddr"中的 s 表示 socket，un 表示 unix，而与"Sun"电脑公司毫无关系。unix_mkname()的代码在文件 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 170 202 ====================

```
170  /*
171   *   Check unix socket name:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
172   *        - should be not zero length.
173   *          - if started by not zero, should be NULL terminated (FS object)
174   *        - if started by zero, it is abstract name.
175   */
176
177  static int unix_mkname(struct sockaddr_un * sunaddr, int len, unsigned *hashp)
178  {
179       if (len <= sizeof(short) || len > sizeof(*sunaddr))
180            return -EINVAL;
181       if (!sunaddr || sunaddr->sun_family != AF_UNIX)
182            return -EINVAL;
183       if (sunaddr->sun_path[0])
184       {
185            /*
186             *   This may look like an off by one error but it is
187             *   a bit more subtle. 108 is the longest valid AF_UNIX
188             *   path for a binding. sun_path[108] doesnt as such
189             *   exist. However in kernel space we are guaranteed that
190             *   it is a valid memory location in our kernel
191             *   address buffer.
192             */
193            if (len > sizeof(*sunaddr))
194                 len = sizeof(*sunaddr);
195            ((char *)sunaddr)[len]=0;
196            len = strlen(sunaddr->sun_path)+1+sizeof(short);
197            return len;
198       }
199
200       *hashp = unix_hash_fold(csum_partial((char*)sunaddr, len, 0));
201       return len;
202  }
```

Unix 域的插口地址有两种类型。一种是常规的路径名字符串，不过不一定以 0 结尾，在长度中也不包括结尾的 0 在内；另一种是以“\0”开头的，称为抽象地址。对于前者，unix_mkname()将其转换成一个以 0 结尾的字符串，并对其长度作出相应调整（195～196 行）。后者类似于网络地址，unix_mkname()为之计算出一个杂凑值，并通过参数 hashp 返回这个杂凑值。

对于这两种不同类型地址的使用和处理是不同的。对常规的字符串地址，unix_bind()根据其路径名为之在文件系统中建立一个“文件”节点。像其它特殊文件一样，这个文件实际上只是一个索引节点，而并没有用于数据的记录块，但是这样一来这个插口在文件系统的目录树中就成为可见的了。以后就可以通过 pathwalk()根据该插口的地址（路径名）在文件系统中找到其 inode，并用索引节点号代替地址的杂凑值来决定将插口的 sock 数据结构挂入杂凑表中的哪个队列。对于“抽象地址”则并不建立这样的文件，所以使用“抽象地址”的 Unix 域插口就像用于其它网域的插口一样，从文件系统的角度来讲是不可见的，并且使用地址的杂凑值来确定挂入到哪个队列中。

回到 unix_bind()的代码中。当插口地址为常规的文件路径名时（677～726 行），首先通过 path_init()和 path_walk()根据给定的路径名找到其父节点，即所在的目录节点，并确认目标节点尚不存在。学习

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

过“文件系统”一章的读者对这些代码不应该有困难。

然后，通过 vfs_mknod()在文件系统的目录树中建立一个文件节点（见 717 行）。这个函数的代码我们在“文件系统”一章中并未触及，所以在这里看一下，它在 fs/namei.c 中：

==================== fs/namei.c 1176 1203 ====================

[sys_socketcall()>sys_bind()>unix_bind()>vfs_mknod()]

```
int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
{
	int error = -EPERM;

	mode &= ~current->fs->umask;

	down(&dir->i_zombie);
	if ((S_ISCHR(mode) || S_ISBLK(mode)) && !capable(CAP_MKNOD))
		goto exit_lock;

	error = may_create(dir, dentry);
	if (error)
		goto exit_lock;

	error = -EPERM;
	if (!dir->i_op || !dir->i_op->mknod)
		goto exit_lock;

	DQUOT_INIT(dir);
	lock_kernel();
	error = dir->i_op->mknod(dir, dentry, mode, dev);
	unlock_kernel();
exit_lock:
	up(&dir->i_zombie);
	if (!error)
		inode_dir_notify(dir, DN_CREATE);
	return error;
}
```

同样，读者对这段代码不应感到困难。操作的主体是由具体文件系统通过其 inode_operations 结构中的函数指针提供的。以 Ext2 文件系统为例，假设路径名决定了目标节点落在 Ext2 文件系统中，或者说其所在目录在 Ext2 文件系统中，那就会调用 Ext2 的 mknod 操作 ext2_mknod()。读者在“文件系统”一章中已经阅读过这个函数的代码。读者应该还记得，在创建插口时已经为之建立了一个 inode 数据结构，而现在 vfs_mknod()显然又会根据插口的路径名在某一文件系统中创建一个索引节点。创建了索引节点以后，当通过系统调用 open()试图打开这个文件时，就会在内存中根据该索引节点的内容建立起一个 inode 数据结构。那么，这样一来插口岂不就有了两个 inode 结构？两个 inode 结构又各有什么作用？我们知道，在打开文件时，是由一个函数 init_special_inodc()根据文件的模式对其 inode 结构进行初始化的，而这里在调用 vfs_mknod()时把文件的模式设置成 S_IFSOCK，所以我们看一下在 init_special_inode()中对插口文件的处理。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/devices.c 200 216 ====================

```
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
     inode->i_mode = mode;
     if (S_ISCHR(mode)) {
          inode->i_fop = &def_chr_fops;
          inode->i_rdev = to_kdev_t(rdev);
     } else if (S_ISBLK(mode)) {
          inode->i_fop = &def_blk_fops;
          inode->i_rdev = to_kdev_t(rdev);
          inode->i_bdev = bdget(rdev);
     } else if (S_ISFIFO(mode))
          inode->i_fop = &def_fifo_fops;
     else if (S_ISSOCK(mode))
          inode->i_fop = &bad_sock_fops;
     else
          printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
}
```

这里的宏操作 S_ISSOCK()检查目标节点的模式是否 S_IFSOCK，如果是就把 inode 结构中的 inode_operations 结构指针 i_fop 设置成指向 bad_sock_fops，这个结构的定义在 fs/devices.c 中：

==================== fs/devices.c 196 198 ====================

```
static struct file_operations bad_sock_fops = {
     open:          sock_no_open
};
```

==================== fs/devices.c 191 194 ====================

```
static int sock_no_open(struct inode *irrelevant, struct file *dontcare)
{
     return -ENXIO;
}
```

显然，这个节点是不允许通过常规的系统调用来打开的。另一方面，插口一经创建就是已打开文件，也不需要通过系统调用 open()来打开。那么，为什么要建立这个文件呢？这是因为 Unix 域允许以常规的路径名作为插口地址，这样的插口地址便于记忆，也便于通过常规的文件操作来检查一个特定的路径名是否已经在使用中。如果文件系统中已经存在具有相同路径名的文件，则 unix_bind()会失败而返回出错代码 EADDRINUSE。所以通常在用户程序中要在调用 bind()之前先调用 unlink()，将可能已经存在的同名文件先删除。应该指出，插口并不是持续存在的，其寿命绝不会超过创建它的进程。当创建插口的进程 exit()时，它所创建的插口也会随着已打开文件的关闭而消失。可是，为插口创建的文件（结点）却是持续存在的；即使创建它的进程已经 exit()，甚至机器已经断电，这文件还是存在于磁盘上，所以必须特地加以删除。再看两个 inode 结构间的关系。我们在前一节中讲过，每个插口的 sock 结构都挂在一个杂凑表中的某个队列里。对于采用路径名为地址的插口，杂凑值是根据文件的索引节点号产生的，所以只要得到了文件的索引节点号，就可以找到插口的 sock 数据结构，从而找到插口的 socket 结构及其宿主，即创建插口时建立的 inode 结构。

当然，Unix 域的插口也可以不用路径名作为地址，那就是前述的“抽象地址”，其第一个字节必须

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

是 0。采用抽象地址的插口在文件系统中不存在插口文件。

如前所述，插口的 sock 结构在创建插口时暂时挂在 unix_socket_table 的最后一个队列中。这个队列的下标超出了杂凑值的范围，所以用杂凑值作下标是不会访问到这个队列里来的。现在既然已经为插口指定了地址，就要把它的 sock 结构转移到相应的队列中了。对于常规的路径名地址，由于所创建文件的 i 节点号码是惟一的，所以不会重复，其 i 节点号码的低 8 位被用作杂凑表中的下标（见 740 行）。从本质上讲，取 i 节点号码的低 8 位也是一种杂凑运算，只不过其杂凑函数特别简单而已。“抽象地址 ”就不同了。在为一个插口指定一个抽象地址时并不能保证这个地址的惟一性，所以要先根据地址的杂凑值在相应的队列中检查一下这个地址是否已经存在。函数__unix_find_socket_byname()的代码在文件 net/unix/af_unix.c 中，很简单：

==================== net/unix/af_unix.c 248 259 ====================
[sys_socketcall()>sys_bind()>unix_bind()>__unix_find_socket_byname()]

```
248  static unix_socket *__unix_find_socket_byname(struct sockaddr_un *sunname,
249                                 int len, int type, unsigned hash)
250  {
251      unix_socket *s;
252
253      for (s=unix_socket_table[hash^type]; s; s=s->next) {
254          if(s->protinfo.af_unix.addr->len==len &&
255             memcmp(s->protinfo.af_unix.addr->name, sunname, len) == 0)
256               return s;
257      }
258      return NULL;
259  }
```

接着，就是把插口的 sock 结构转移到已经确定的队列中了（见 746 行和 748 行）。

最后，回到 sys_bind()的代码中，sockfd_put()的作用是递减对插口的 file 结构的使用计数。这是当初在 sockfd_lookup()中通过 fget()递增的，所以递减以后不会达到 0。

7.4 函数 sys_listen()——设定 server 插口

以前讲过，“有连接”模式的插口天生就是按 client/server 的模式运转的，只有在一个 server 插口和一个 client 插口之间才能建立起连接。那么，怎样来区分这两种不同的插口呢？只要在插口创建以后为其调用了 listen()，这个插口就成为 server 插口了。凡是 server 插口都不能主动去与别的插口建立连接，而只能被动地通过 accept()接受来自 client 插口的连接请求。而 client 插口则相反，不能调用 accept()来接受连接请求，而只能主动地通过 connect()提出连接请求。

我们来看 sys_listen()的代码（net/socket.c）：

==================== net/socket.c 1003 1019 ====================
[sys_socketcall()>sys_listen()]

```
1003  *   necessary for a listen, and if that works, we mark the socket as
1004  *   ready for listening.
1005  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

asmlinkage long sys_listen(int fd, int backlog)
{
        struct socket *sock;
        int err;

        if ((sock = sockfd_lookup(fd, &err)) != NULL) {
                if ((unsigned) backlog > SOMAXCONN)
                        backlog = SOMAXCONN;
                err=sock->ops->listen(sock, backlog);
                sockfd_put(sock);
        }
        return err;
}


```

与 sys_bind()的代码比较一下就可以看出，除对参数 backlog 的处理外，二者基本上是一样的，都是根据打开文件号 fd 找到插口的 socket 数据结构（inode 的一部分），进而通过结构中的指针 ops 找到相应的 proto_ops 数据结构 unix_stream_ops 或 unix_dgram_ops，再通过不同的函数指针执行相应的函数。数据结构 unix_dgram_ops 中的指针 listen 设置为 sock_no_listen()，说明无连接模式的插口并不支持 listen()，而 unix_stream_ops 中的指针 listem 则设置为 unix_listen()，此函数在文件 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 431 459 ====================
[sys_socketcall()>sys_listen()>unix_listen()]

```
static int unix_listen(struct socket *sock, int backlog)
{
        int err;
        struct sock *sk = sock->sk;

        err = -EOPNOTSUPP;
        if (sock->type!=SOCK_STREAM)
                goto out;           /* Only stream sockets accept */
        err = -EINVAL;
        if (!sk->protinfo.af_unix.addr)
                goto out;           /* No listens on an unbound socket */
        unix_state_wlock(sk);
        if (sk->state != TCP_CLOSE && sk->state != TCP_LISTEN)
                goto out_unlock;
        if (backlog > sk->max_ack_backlog)
                wake_up_interruptible_all(&sk->protinfo.af_unix.peer_wait);
        sk->max_ack_backlog=backlog;
        sk->state=TCP_LISTEN;
        /* set credentials so connect can copy them */
        sk->peercred.pid = current->pid;
        sk->peercred.uid = current->euid;
        sk->peercred.gid = current->egid;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
453         err = 0;
454
455     out_unlock:
456         unix_state_wunlock(sk);
457     out:
458         return err;
459     }
```

首先，只有插口的类型为 SOCK_STREAM，即“有连接”模式的插口，并且已经为其 bind()了插口地址，才允许 listen()。其次，对于符合这些条件的插口也不是什么时候都可以调用 listen()的。插口的 sock 结构中有个成分 state，用来实现一种“有限状态机”。只有当这个状态机处于 TCP_CLOSE 或 TCP_LISTEN 这两种状态时才可以对其调用 listen()。在前面 seek_create()的代码中，我们看到在创建一个插口时要调用函数 sock_init_data()对分配的 sock 数据结构进行初始化，在那里将 state 设置成 TCP_CLOSE。状态 TCP_CLOSE 表示插口只是刚刚建立，尚未宣称为 server 插口；而 TCP_LISTEN 则表示该插口已经设置成 server 插口，但尚未建立起连接，并且不是在等待来自 client 一方的连接请求。只有在这两种状态下才允许改变插口的参数（主要是连接请求队列的容量）。这里顺便要提一下，“有连接”模式的插口未必都是采用 TCP 规程的，但 TCP 是一种典型的“有连接”规程，所以 sock 结构中实现的是一个 TCP 状态机的子集。这样，即使插口的具体规程并不是 TCP，从插口的使用来说已经能满足要求了。

至于 listen()真正要做的事情，其实是相当简单的（见 447～452 行）。除状态本身的变化以外，就是设置（或改变）几个参数，主要就是最大队列长度 max_ack_backlog，还有就是“插口主”的身份，即所谓 credentials，具体就是进程的用户号、组号以及进程号。以后，当 server 方接受一个连接请求时，要将这些信息送回给请求连接的一方，让其知道究竟是谁接受了连接请求。还有，当新的队列容量比原来有所扩大时，还要唤醒可能正在睡眠中等待着将连接请求挂入队列的进程（445～446 行）。

7.5　函数 sys_accept()——接受连接请求

“有连接”模式的插口一旦通过 listen()设置成 server 插口以后，就只能被动地通过 accept()接受来自 client 插口的连接请求。进程对 accept()的调用是阻塞性的，就是说如果没有连接请求就会进入睡眠等待，直到有连接请求到来，接受了请求以后（或者超过了预定的等待时间）才会返回。所以，在已经有连接请求的情况下是“接受连接请求”，而在尚无连接请求的情况下是“等待连接请求”。在内核中，accept()是通过 net/socket.c 中的函数 sys_accept()实现的，其代码在 net/socket.c 中：

==================== net/socket.c 1022 1078 ====================

[sys_socketcall()>sys_accept()]

```
/*
 *   For accept, we attempt to create a new socket, set up the link
 *   with the client, wake up the client, then return the new
 *   connected fd. We collect the address of the connector in kernel
 *   space and move it to user at the very end. This is unclean because
 *   we open the socket then return an error.
 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *	1003.1g adds the ability to recvmsg() to query connection pending
 *	status to recvmsg. We need to add that support in a way thats
 *	clean when we restucture accept also.
 */

asmlinkage long sys_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
{
	struct socket *sock, *newsock;
	int err, len;
	char address[MAX_SOCK_ADDR];

	sock = sockfd_lookup(fd, &err);
	if (!sock)
		goto out;

	err = -EMFILE;
	if (!(newsock = sock_alloc()))
		goto out_put;

	newsock->type = sock->type;
	newsock->ops = sock->ops;

	err = sock->ops->accept(sock, newsock, sock->file->f_flags);
	if (err < 0)
		goto out_release;

	if (upeer_sockaddr) {
		if(newsock->ops->getname(newsock, (struct sockaddr *)address, &len, 2)<0) {
			err = -ECONNABORTED;
			goto out_release;
		}
		err = move_addr_to_user(address, len, upeer_sockaddr, upeer_addrlen);
		if (err < 0)
			goto out_release;
	}

	/* File flags are not inherited via accept() unlike another OSes. */

	if ((err = sock_map_fd(newsock)) < 0)
		goto out_release;

out_put:
	sockfd_put(sock);
out:
	return err;

out_release:
	sock_release(newsock);
	goto out_put;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

说来也许奇怪，实际上一个插口经 listen()设置成 server 插口以后永远不会与任何插口建立起连接。因为一旦接受了一个连接请求之后就会创建出另一个插口，连接就建立在这个新的插口与请求连接的那个插口之间，而原先的 server 插口则并无改变，并且还可再次通过 accept()接受下一个连接请求，就好像母鸡下蛋一样。就这样“只取蛋，不杀鸡”，server 插口永远保持接受新的连接请求的能力，但是其本身从来不成为某个连接的一端。正因为这样，sys_accept()返回的是新插口的打开文件号，同时还通过作为参数的指针 upeer_sockaddr 返回对方的插口地址。

代码中先通过 sockfd_lookup()找到 server 插口的 socket 结构，然后通过 sock_alloc()分配一个新的 socket 结构，为“下蛋”作好准备；再通过相应 proto_ops 数据结构（对于 Unix 域是 unix_stream_ops 或 unix_dgram_ops）中的函数指针 accept 调用具体的函数。从前面数据结构 unix_dgram_ops 的代码可以看到，“无连接”插口的 accept 函数指针设置为 sock_no_accept()，可见“无连接”插口并不支持 accept()。而“有连接”插口的 accept 指针则指向 unix_accept()，此函数在文件 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1038 1074 ====================

[sys_socketcall()>sys_accept()>unix_accept()]

```
static int unix_accept(struct socket *sock, struct socket *newsock, int flags)
{
	unix_socket *sk = sock->sk;
	unix_socket *tsk;
	struct sk_buff *skb;
	int err;

	err = -EOPNOTSUPP;
	if (sock->type!=SOCK_STREAM)
		goto out;

	err = -EINVAL;
	if (sk->state!=TCP_LISTEN)
		goto out;

	/* If socket state is TCP_LISTEN it cannot change (for now...),
	 * so that no locks are necessary.
	 */

	skb = skb_recv_datagram(sk, 0, flags&O_NONBLOCK, &err);
	if (!skb)
		goto out;

	tsk = skb->sk;
	skb_free_datagram(sk, skb);
	wake_up_interruptible(&sk->protinfo.af_unix.peer_wait);

	/* attach accepted sock to socket */
	unix_state_wlock(tsk);
	newsock->state = SS_CONNECTED;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        sock_graft(tsk, newsock);
        unix_state_wunlock(tsk);
        return 0;

out:
        return err;
}
```

以前我们在讲到插口的创建时曾经提到，在 sock 结构中有几个重要的队列，其中之一是到达报文的队列 receive_queue。到达的报文（更确切地说是 packet）“载运”在 sk_buff 数据结构中，而 receive_queue 等待队列就是 sk_buff 结构的队列。连接请求也是以报文的形式到达的，不过不是一般的数据报文，而属于控制报文。所以，所谓等待连接请求的到来就是等待这种控制报文的到来。另一方面，虽然这里所说插口都是“有连接”模式的，但那只是对数据（报文）而言，而控制报文实际上都是“无连接”的（否则，靠什么手段来建立最初的连接呢？）所以，这里通过 skb_recv_datagram()从 receive_queue 队列中接收代表着连接请求的控制报文。注意 datagram（“数据报”）并不表示“数据报文”，而是说以“无连接”模式传递的报文，切不要把这二者混淆了。

函数 skb_recv_datagram()的代码在 net/core/datagram.c 中：

==================== net/core/datagram.c 109 179 ====================

[sys_socketcall()>sys_accept()>unix_accept()>skb_recv_datagram()]

```
/*
 *    Get a datagram skbuff, understands the peeking, nonblocking wakeups and possible
 *    races. This replaces identical code in packet,raw and udp, as well as the IPX
 *    AX.25 and Appletalk. It also finally fixes the long standing peek and read
 *    race for datagram sockets. If you alter this routine remember it must be
 *    re-entrant.
 *
 *    This function will lock the socket if a skb is returned, so the caller
 *    needs to unlock the socket in that case (usually by calling skb_free_datagram)
 *
 *    * It does not lock socket since today. This function is
 *    * free of race conditions. This measure should/can improve
 *    * significantly datagram socket latencies at high loads,
 *    * when data copying to user space takes lots of time.
 *    * (BTW I've just killed the last cli() in IP/IPv6/core/netlink/packet
 *    *  8) Great win.)
 *    *                               --ANK (980729)
 *
 *    The order of the tests when we find no data waiting are specified
 *    quite explicitly by POSIX 1003.1g, don't change them without having
 *    the standard around please.
 */

struct sk_buff *skb_recv_datagram(struct sock *sk, unsigned flags, int noblock, int *err)
{
        int error;
        struct sk_buff *skb;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	long timeo;

	/* Caller is allowed not to check sk->err before skb_recv_datagram() */
	error = sock_error(sk);
	if (error)
		goto no_packet;

	timeo = sock_rcvtimeo(sk, noblock);

	do {
		/* Again only user level code calls this function, so nothing interrupt level
		   will suddenly eat the receive_queue.

		   Look at current nfs client by the way...
		   However, this function was corrent in any case. 8)
		 */
		if (flags & MSG_PEEK)
		{
			unsigned long cpu_flags;

			spin_lock_irqsave(&sk->receive_queue.lock, cpu_flags);
			skb = skb_peek(&sk->receive_queue);
			if(skb!=NULL)
				atomic_inc(&skb->users);
			spin_unlock_irqrestore(&sk->receive_queue.lock, cpu_flags);
		} else
			skb = skb_dequeue(&sk->receive_queue);

		if (skb)
			return skb;

		/* User doesn't want to wait */
		error = -EAGAIN;
		if (!timeo)
			goto no_packet;

	} while (wait_for_packet(sk, err, &timeo) == 0);

	return NULL;

no_packet:
	*err = error;
	return NULL;
}
```

其实我们应该在讲述“无连接”模式的报文接收时再来介绍这个函数，但是既然在这儿先碰上了，也就只好把它提前了。

首先是检查 sock 结构中的出错代码 err，看看从上一次调用这个函数以后至今是否发生了出错，同

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

时将其清 0。这里的 sock_error()是个 inline 函数，其定义在 include/net/sock.h 中：

==================== include/net/sock.h 1197 1205 ====================
[sys_socketcall()>sys_accept()>unix_accept()>skb_recv_datagram()>sock_error()]

```
/*
 *   Recover an error report and clear atomically
 */

static inline int sock_error(struct sock *sk)
{
     int err=xchg(&sk->err,0);
     return -err;
}
```

如果没有出错的话，就可以试图从队列中接收（脱链）一个报文（即 sk_buff 数据结构）了。这里的标志位 MSG_PEEK 是为 recv()、recvfrom()等库函数设置的。在调用那些库函数时可以把参数 flags 中的 MSG_PEEK 置成 1，表示只是看一下队列中是否有报文可接收，但是并不真的接收。除此之外，还有个在文件操作中常用的标志位 O_NONBLOCK，表示如果有报文就接收，但若没有也得马上返回，而不是睡眠等待，这个标志位也可用于 accept()。进一步，还可以为等待连接的操作设定一个时间限制，这就是参数 noblock 的作用。

大家都知道，队列操作是绝不容许打扰的（这种打扰可能来自其它进程，也有可能来自中断服务程序），所以在 skb_peek()之前要加锁。至于 skb_dequeue()，则已经把这一点考虑进去了（见 include/linux/skbuff.h）：

==================== include/linux/skbuff.h 513 531 ====================
[sys_socketcall()>sys_accept()>unix_accept()>skb_recv_datagram()>skb_dequeue()]

```
/**
 *   skb_dequeue - remove from the head of the queue
 *   @list: list to dequeue from
 *
 *   Remove the head of the list. The list lock is taken so the function
 *   may be used safely with other locking list functions. The head item is
 *   returned or %NULL if the list is empty.
 */

static inline struct sk_buff *skb_dequeue(struct sk_buff_head *list)
{
     long flags;
     struct sk_buff *result;

     spin_lock_irqsave(&list->lock, flags);
     result = __skb_dequeue(list);
     spin_unlock_irqrestore(&list->lock, flags);
     return result;
}
```

显然，其主体是__skb_dequeue()：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/skbuff.h 484 511 ====================

[sys_socketcall()>sys_accept()>unix_accept()>skb_recv_datagram()>skb_dequeue()>__skb_dequeue()]

```
/**
 *   __skb_dequeue - remove from the head of the queue
 *   @list: list to dequeue from
 *
 *   Remove the head of the list. This function does not take any locks
 *   so must be used with appropriate locks held only. The head item is
 *   returned or %NULL if the list is empty.
 */

static inline struct sk_buff *__skb_dequeue(struct sk_buff_head *list)
{
     struct sk_buff *next, *prev, *result;

     prev = (struct sk_buff *) list;
     next = prev->next;
     result = NULL;
     if (next != prev) {
          result = next;
          next = next->next;
          list->qlen--;
          next->prev = prev;
          prev->next = next;
          result->next = NULL;
          result->prev = NULL;
          result->list = NULL;
     }
     return result;
}
```

如果队列中有报文的话，接收成功，就可以返回了（见 165 行）。要是没有呢？那就要看在调用 accept()时的 O_NONBLOCK 标志是否为 1，如果是的话，就马上出错返回（见 170 行），出错代码为-EAGAIN；否则就通过 wait_for_packet()睡眠等待。这个函数的代码在 net/core/datagram.c 中：

==================== net/core/datagram.c 63 107 ====================

[sys_socketcall()>sys_accept()>unix_accept()>skb_recv_datagram()>wait_for_packet()]

```
static int wait_for_packet(struct sock * sk, int *err, long *timeo_p)
{
     int error;

     DECLARE_WAITQUEUE(wait, current);

     __set_current_state(TASK_INTERRUPTIBLE);
     add_wait_queue_exclusive(sk->sleep, &wait);

     /* Socket errors? */
     error = sock_error(sk);
     if (error)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            goto out;

        if (!skb_queue_empty(&sk->receive_queue))
            goto ready;

        /* Socket shut down? */
        if (sk->shutdown & RCV_SHUTDOWN)
            goto out;

        /* Sequenced packets can come disconnected. If so we report the problem */
        error = -ENOTCONN;
        if(connection_based(sk) && !(sk->state==TCP_ESTABLISHED || sk->state==TCP_LISTEN))
            goto out;

        /* handle signals */
        if (signal_pending(current))
            goto interrupted;

        *timeo_p = schedule_timeout(*timeo_p);

    ready:
        current->state = TASK_RUNNING;
        remove_wait_queue(sk->sleep, &wait);
        return 0;

    interrupted:
        error = sock_intr_errno(*timeo_p);
    out:
        current->state = TASK_RUNNING;
        remove_wait_queue(sk->sleep, &wait);
        *err = error;
        return error;
    }
```

这里 sock 数据结构中的指针 sleep 指向 socket 结构中的队列 wait，是创建插口时在 sock_init_data()中设置好了的。至于数据结构 wait，是由 DECLARE_WAITQUEUE()定义的一个局部量，其空间在当前进程的系统空间堆栈上，也就是在 wait_for_packet()的调用框架中，只要不从这个函数返回就一直是有效的。通过 wait_for_packet()将数据结构 wait 挂入 sock 结构的 sleep 队列，就等于把当前进程挂入了目标插口的（连接请求）等待队列。从这里也可看出，多个进程在同一个插口上等待连接请求是允许的。当前进程通过 schedule_timeout()进入睡眠以后，就一直要到有下列事件之一发生时才会被唤醒而从 schedule_timeout()返回，那就是：①有连接请求到达；②接收到了一个信号；③到达了预定的等待时间。在这三种情况下 wait_for_packet()的返回值都是 0，所以在 skb_recv_datagram()的 do-while 循环中都会再执行一次循环体，再作最后一次尝试。此外，wait_for_packet()还通过指针 timeo_p 返回还剩下的等待时间。

读者也许注意到了，这里的处理与进程在许多系统调用中从睡眠醒来时有所不同。一般，当一个进程从睡眠中醒来时要先检查是否有信号到达，如果有就提前结束本次系统调用，先对信号作出反应。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

而这里则不同，醒来后还是回到循环体中再试一次，这是为什么呢？实际上，是否要提前结束本次系统调用不能一概而论，要看继续往下执行是否能在一个有限的短时间内完成。在这里的循环体中执行的基本上就是 skb_dequeue()，那只是举手之劳，下面还会看到在接收到一个报文后的处理也很简单，所以还不如再试一次，哪怕不成功，再来处理信号也还不迟。所以，如果这一次成功了当然最好，那就会在 165 行返回；要是仍不成功，而时间已经到点（timeo 变成了 0），那就会在 170 行结束循环；否则，那就应该是因为接收到信号而被唤醒了（队列中没有报文，时间又未到点），当再次调用 wait_for_packet()的时候就会在那里的 91 行转到 interrupted 处，从而结束 do-while 循环并返回一个出错代码。至于对信号的处理，则在从系统调用返回时自会按常规进行。

有关连接请求的到达，以及唤醒正在 sys_accept()中睡眠的进程可以参看后面对 connect()的介绍，这里我们假定连接请求已经到达了。所以，从 wait_for_packet()返回以后再试一次就成功了。

回到 unix_accept()的代码中（1058 行），指针 skb 指向接收到的 sk_buff 数据结构。这种数据结构是在 include/linux/skbuff.h 中定义的，由于要考虑到各种不同的规程，其定义长达将近 100 行。另一方面，我们在本书中所关心的只是 Unix 域，所以就不列出它的定义了。对于 Unix 域的插口，在 sk_buff 数据结构中有个 sock 结构指针 sk，指向一个 sock 数据结构，就是代码中的 tsk。这个 sock 结构是由请求连接的那一方送过来、供接受连接的一方使用的，注意 tsk 与“ task”毫无关系，大概是“ target sock ”的意思。

前面，在 sys_accept()中（1045 行）已经调用，sock_alloc()，分配了一个新的 socket 结构（也就是 inode 结构），那就是 unix_accept()的参数 newsock。但 sock_alloc()并不是 sock_create()，它并不分配与 socket 结构配对的 sock 结构，从这个意义上讲，这个新的插口还不完整。那么什么时候才分配所需的 sock 数据结构呢？这是由 client 一方在 connect()的过程中分配，并且将其指针通过用作连接请求报文的 sk_buff 结构带过来的，这就是这里的 tsk。现在，就通过 sock_graft()将 client 一方提供的 tsk 与 server 一方提供的 newsock 挂上钩（1068 行）。

==================== include/net/sock.h 1014 1021 ====================

[sys_socketcall()>sys_accept()>unix_accept()>sock_graft()]

```
static inline void sock_graft(struct sock *sk, struct socket *parent)
{
	write_lock_bh(&sk->callback_lock);
	sk->sleep = &parent->wait;
	parent->sk = sk;
	sk->socket = parent;
	write_unlock_bh(&sk->callback_lock);
}
```

这样，就完成了主要的连接过程，因为 tsk 与 client 一方的 sock 结构事先已经互相“背靠背”连接好了。同时，新的 socket 结构的状态也直接设置成 SS_CONNECTED。但是请注意，原先的 socket 结构的状态却并未改变。

读者也许会问，代码中并没有对接收到的报文，即 sk_buff 结构，作什么检查，怎么就能知道它一定是个连接请求报文呢？答案是，只有连接请求才能挂入到 server 插口的 receive_queue 队列中。数据报文是不能够挂入到这个队列中的，它们只能挂入到建立了连接的新插口的 receive_queue 队列中。另外，也只有在 connect()的过程中才会唤醒正在等待连接请求的进程。所以，对于 Unix 域的插口来说，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

是否有“控制报文”这个概念，其实也无关紧要，只是为了与网络环境下的插口机制保持概念上以及实现上的一致性才套用了这些概念。

接受了一个连接请求以后，一方面要释放报文，另一方面要把请求连接的一方唤醒。前面提到过，receive_queue 队列的长度是有限制的，如果队列中的报文数量已经达到了上限，则新到达的报文就因无法“投递”而只好让 client 方进入睡眠等待。现在，server 方接收了一个报文以后，receive_queue 队列的长度下降了，就可以把可能正在睡眠等待的进程唤醒了（1063 行）。这里的 wake_up_interruptible() 是个宏操作，定义于 include/linux/sched.h 中：

==================== include/linux/sched.h 558 558 ====================

```
#define wake_up_interruptible(x) __wake_up((x),TASK_INTERRUPTIBLE,WQ_FLAG_EXCLUSIVE)
```

调用参数 WQ_FLAG_EXCLUSIVE 表示如果有多个进程在睡眠等待就只唤醒其中一个。

回到 sys_accept() 的代码中。如果用户进程提供了用来返回对方地址的数据结构指针 upeer_sockaddr，就要通过相应 proto_ops 结构中的函数指针 getname 来获取对方的插口地址。在数据结构 unix_stream_ops 中，函数指针 getname 设置为 unix_getname()，其代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1077 1109 ====================

[sys_socketcall()>sys_accept()>unix_getname()]

```
static int unix_getname(struct socket *sock, struct sockaddr *uaddr, int *uaddr_len, int peer)
{
    struct sock *sk = sock->sk;
    struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
    int err = 0;

    if (peer) {
        sk = unix_peer_get(sk);

        err = -ENOTCONN;
        if (!sk)
            goto out;
        err = 0;
    } else {
        sock_hold(sk);
    }

    unix_state_rlock(sk);
    if (!sk->protinfo.af_unix.addr) {
        sunaddr->sun_family = AF_UNIX;
        sunaddr->sun_path[0] = 0;
        *uaddr_len = sizeof(short);
    } else {
        struct unix_address *addr = sk->protinfo.af_unix.addr;

        *uaddr_len = addr->len;
        memcpy(sunaddr, addr->name, *uaddr_len);
    }
    unix_state_runlock(sk);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    sock_put(sk);
out:
    return err;
}
```

这个函数既可以用来获取对方插口的地址，也可以用来换取本插口的地址，具体由参数 peer 决定。在这里由于调用时将参数的值设成 1，所以取的是对方地址。函数 unix_peer_get()通过 sock 结构中的指针 pair 取得指向对方 sock 结构的指针。如前所述，与 newsock 配对的 sock 结构是由调用 connect()的一方分配和设置的，当然也包括指针 pair 的设置。所以，在调用了 unix_peer_get()以后，代码中的指针 sk 就改成指向对方的 sock 结构了。

==================== net/unix/af_unix.c 152 162 ====================
[sys_socketcall()>sys_accept()>unix_getname()>unix_peer_get()]

```
static __inline__ unix_socket * unix_peer_get(unix_socket *s)
{
    unix_socket *peer;

    unix_state_rlock(s);
    peer = unix_peer(s);
    if (peer)
        sock_hold(peer);
    unix_state_runlock(s);
    return peer;
}
```

==================== net/unix/af_unix.c 140 140 ====================

```
#define unix_peer(sk) ((sk)->pair)反之，sock_hold()
```

反之，sock_hold()则只是递增 sock 结构中的使用计数（见 include/net/sock.h）：

==================== include/net/sock.h 972 981 ====================
[sys_socketcall()>sys_accept()>unix_getname()>sock_hold()]

```
/* Grab socket reference count. This operation is valid only
   when sk is ALREADY grabbed f.e. it is found in hash table
   or a list and the lookup is made under lock preventing hash table
   modifications.
 */

static inline void sock_hold(struct sock *sk)
{
    atomic_inc(&sk->refcnt);
}
```

前面讲过，server 插口必须有个地址，这样 client 一方才能通过地址来找到 server 插口的数据结构，所以在创建了插口以后一定要调用 bind()将其“捆绑”到一个地址上。可是，对于 client 一方的插口来说，它只能主动地去寻找某个 server 插口，别的插口是不会来寻找它的。尤其是 Unix 域的插口，所传递的报文都是在同一计算机中而无需穿越网络。所以实际上没有必要让 client 插口也有个地址，当然有

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

也无妨。这样，在 unix_getname()中就要考虑到两种情况，一种是 sock 结构所属的插口根本就没有地址（1095 行），另一种则是有地址的（1099 行）。

再回到 sys_accept()的代码中。最后一件事就是为新创建的插口也分配一个打开文件号以及相应的 file 结构，并返回这个打开文件号。函数 sock_map_fd()的代码读者在 sys_socket()中已经看到过，这里就不重复了。

7.6 函数 sys_connect()——请求连接

以前讲过，“有连接”模式的插口与“无连接”模式的插口都可以调用库函数 connect()，但是意义却不同。内核中的函数 sys_connect()是二者公用的，只是因“安装”在插口上的 proto_ops 数据结构的不同而通过不同的函数指针转入不同的处理程序。函数 sys_connect()的代码在文件 net/socket.c 中：

==================== net/socket.c 1081 1111 ====================

[sys_socketcall()>sys_connect()]

```
1081 /*
1082  *   Attempt to connect to a socket with the server address.  The address
1083  *   is in user space so we verify it is OK and move it to kernel space.
1084  *
1085  *   For 1003.1g we need to add clean support for a bind to AF_UNSPEC to
1086  *   break bindings
1087  *
1088  *   NOTE: 1003.1g draft 6.3 is broken with respect to AX.25/NetROM and
1089  *   other SEQPACKET protocols that take time to connect() as it doesn't
1090  *   include the -EINPROGRESS status for such sockets.
1091  */
1092
1093 asmlinkage long sys_connect(int fd, struct sockaddr *uservaddr, int addrlen)
1094 {
1095     struct socket *sock;
1096     char address[MAX_SOCK_ADDR];
1097     int err;
1098
1099     sock = sockfd_lookup(fd, &err);
1100     if (!sock)
1101         goto out;
1102     err = move_addr_to_kernel(uservaddr, addrlen, address);
1103     if (err < 0)
1104         goto out_put;
1105     err = sock->ops->connect(sock, (struct sockaddr *) address, addrlen,
1106                  sock->file->f_flags);
1107 out_put:
1108     sockfd_put(sock);
1109 out:
1110     return err;
1111 }
```

如前所述，socket 结构中的指针 ops 指向某个 proto_ops 数据结构，对于 unix 域的插口有两种 proto_ops 数据结构，分别用于“有连接”插口和“无连接”插口。数据结构 unis_stream_ops 中的指针 connect 指向 unix_stream_connect()，而 unix_dgram_ops 中的这个指针则指向 unix_dgram_connect()。

先来看“有连接”模式的插口。如前所述，只有 client 插口才可以（并且一定要）通过 connect() 向一个 server 插口提出连接请求，在请求被 server 插口接受而建立起连接之前是不能在两个插口之间传递数据报文的。函数 unix_stream_connect()的代码在 net/unix/af_unix.c 中，我们分段往下看：

==================== net/unix/af_unix.c 852 892 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()]

```
static int unix_stream_connect(struct socket *sock, struct sockaddr *uaddr,
                    int addr_len, int flags)
{
     struct sockaddr_un *sunaddr=(struct sockaddr_un *)uaddr;
     struct sock *sk = sock->sk;
     struct sock *newsk = NULL;
     unix_socket *other = NULL;
     struct sk_buff *skb = NULL;
     unsigned hash;
     int st;
     int err;
     long timeo;

     err = unix_mkname(sunaddr, addr_len, &hash);
     if (err < 0)
          goto out;
     addr_len = err;

     if (sock->passcred && !sk->protinfo.af_unix.addr &&
         (err = unix_autobind(sock)) != 0)
          goto out;

     timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);

     /* First of all allocate resources.
        If we will make it after state is locked,
        we will have to recheck all again in any case.
      */

     err = -ENOMEM;

     /* create new sock for complete connection */
     newsk = unix_create1(NULL);
     if (newsk == NULL)
          goto out;

     /* Allocate skb for sending to listening sock */
     skb = sock_wmalloc(newsk, 1, 0, GFP_KERNEL);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
890      if (skb == NULL)
891          goto out;
892
```

这里函数 unix_mkname()的作用是将目标插口的地址加以某种“规格化”并返回其长度。如果这个地址是一个“抽象地址”，则还要计算出它的杂凑值，其代码我们已经在介绍 sys_socket()时看到过了。插口的 socket 数据结构中有个标志 passcred，意为“ pass credentials”，就是要把自己的“身份”告诉对方。在插口创建之初这个标志为 0，但是可以在用户程序中通过 setsockopt()将其设置成 1。所谓身份，当然包括地址。所以，如果插口没有地址却又得把身份告诉对方，则必须调用 unix_autobind()自动生成一个地址（871 行）。

与 accept()一样，connect()也是阻塞性的，如果连接请求不能得到 server 方接受就会进入睡眠等待，直到 server 方接受了连接请求（或者超过了预定的等待时间）才会返回。不过也可以设置参数 flags 中的 O_NONBLOCK 标志位，使得在连接请求不能马上得到接受时就立即返回。这里的 sock_sndtimeo()是个 inline 函数，定义于 include/net/sock.h 中：

==================== include/net/sock.h 1249 1252 ====================
[sys_socketcall()>sys_connect()>unix_stream_connect()>sock+sndtimeo()]

```
static inline long sock_sndtimeo(struct sock *sk, int noblock)
{
    return noblock ? 0 : sk->sndtimeo;
}
```

然后，通过 unix_create1()分配一个新的 sock 数据结构，其代码我们已经在介绍 sys_socket()时看到过。这个新的 sock 数据结构是为 server 一方在调用 accept()时创建新的插口而准备的。如前所述，一个插口除了有一个 socket 数据结构（实际上是 inode 数据结构的一部分）外，还要有个与之配套使用的 sock 结构。对 Unix 域的“有连接”模式插口而言，这个数据结构是由 client 一方在这里分配的，并将其地址 newsk 通过连接请求报文传递到 server 一方。代表着连接请求的 sk_buff 数据结构则由 sock_wmalloc()分配并初始化（net/core/sock.c）：

==================== net/core/sock.c 654 667 ====================
[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_wmalloc()]

```
/*
 * Allocate a skb from the socket's send buffer.
 */
struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force, int priority)
{
    if (force || atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
        struct sk_buff * skb = alloc_skb(size, priority);
        if (skb) {
            skb_set_owner_w(skb, sk);
            return skb;
        }
    }
    return NULL;
}
```

代码中的 sk->wmem_alloc 是 sock 结构中的一个计数器，用来累计为本插口分配的 sk_buff 存储空间，一般不应超过为该插口设置的限额 sk->sndbuf，但是可以通过把参数 force 设成 1 来打破这个限制。参数 size 为随同 sk_buff 结构发送的数据缓冲区的大小，这里在调用时设置成 1，不过实际分配时是以 16 个字节为单位的，所以实际上是 16 个字节。函数 skb_owner_w()对分配到的 sk_buff 进行一些初始化（include/net/sock.h）：

==================== include/net/sock.h 1125 1140 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_wmalloc()>skb_owner_w()]

```
/*
 *   Queue a received datagram if it will fit. Stream and sequenced
 *   protocols can't normally use this as they need to fit buffers in
 *   and play with them.
 *
 *   Inlined as it's very short and called for pretty much every
 *   packet ever received.
 */

static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
{
     sock_hold(sk);
     skb->sk = sk;
     skb->destructor = sock_wfree;
     atomic_add(skb->truesize, &sk->wmem_alloc);
}
```

这里使报文缓冲区中的指针 sk 指向为 server 方分配的 sock 数据结构，同时还要把报文缓冲区中的函数指针 destructor 设置成指向 sock_wfree()，使 server 方将来能按正确的途径释放这个缓冲区。最后，skb->truesize 为包括数据缓冲区在内的实际大小，其初始值是在 alloc_skb()中设置的。

回到 unix_stream_connect()中。至此，我们已经分配了一个 sock 结构以及一个空白的 sk_huff 结构。让我们继续往下看（net/unix/af_unix.c）：

==================== net/unix/af_unix.c 893 926 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()]

```
restart:
     /*  Find listening sock. */
     other=unix_find_other(sunaddr, addr_len, sk->type, hash, &err);
     if (!other)
          goto out;

     /* Latch state of peer */
     unix_state_rlock(other);

     /* Apparently VFS overslept socket death. Retry. */
     if (other->dead) {
          unix_state_runlock(other);
          sock_put(other);
          goto restart;
```

```
    }

    err = -ECONNREFUSED;
    if (other->state != TCP_LISTEN)
        goto out_unlock;

    if (skb_queue_len(&other->receive_queue) > other->max_ack_backlog) {
        err = -EAGAIN;
        if (!timeo)
            goto out_unlock;

        timeo = unix_wait_for_peer(other, timeo);

        err = sock_intr_errno(timeo);
        if (signal_pending(current))
            goto out;
        sock_put(other);
        goto restart;
    }

```

函数 unix_find_other()根据给定的地址找到目标插口的 sock 数据结构，其代码也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 589 633 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>unix_find_other()]

```
static unix_socket *unix_find_other(struct sockaddr_un *sunname, int len,
                                    int type, unsigned hash, int *error)
{
    unix_socket *u;
    struct nameidata nd;
    int err = 0;

    if (sunname->sun_path[0]) {
        if (path_init(sunname->sun_path,
                      LOOKUP_POSITIVE|LOOKUP_FOLLOW, &nd))
            err = path_walk(sunname->sun_path, &nd);
        if (err)
            goto fail;
        err = permission(nd.dentry->d_inode,MAY_WRITE);
        if (err)
            goto put_fail;

        err = -ECONNREFUSED;
        if (!S_ISSOCK(nd.dentry->d_inode->i_mode))
            goto put_fail;
        u=unix_find_socket_byinode(nd.dentry->d_inode);
        if (!u)
            goto put_fail;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

        path_release(&nd);

        err=-EPROTOTYPE;
        if (u->type != type) {
            sock_put(u);
            goto fail;
        }
    } else {
        err = -ECONNREFUSED;
        u=unix_find_socket_byname(sunname, len, type, hash);
        if (!u)
            goto fail;
    }
    return u;

put_fail:
    path_release(&nd);
fail:
    *error=err;
    return NULL;
}
```

可见，对于常规的以文件路径名为代表的插口地址，要先通过文件系统的操作 path_init()和 path_walk()在文件系统中找到其目录项和索引节点，并在内存中建立起相应的 dentry 结构以及 inode 结构。然后，就使用其索引节点号，通过 unix_find_sock_byinode()在杂凑表 unix_socket_table 中找到相应的队列和 sock 数据结构。对于“抽象地址”，则使用该地址的杂凑值通过 unix_find_socket_byname()在 unix_socket_table 中寻找。找到了对方的 sock 结构以后，二者都会通过 sock_hold()将结构中的访问计数 refcnt 加 1，表示这个结构现在多了一个“用户”。

回到 unix_stream_connect()的代码中（895 行）。找到了 server 插口的 sock 结构以后，指针 other 指向这个结构。但是，在对这个 sock 结构进行任何操作之前需要确认这个 sock 结构不处于正被撤销的过程中。虽然对 Unix 域的插口而言，server 和 client 双方都在同一台计算机上，但是在 SMP 多处理器结构下两个进程有可能在两个不同的处理器中运行（但是共用内存）。所以有可能发生这么一种情况，就是当 client 方的进程运行到这里，即从杂凑表的队列中找到了 server 插口的 sock 结构时，恰好在另一个处理器中运行的 server 方的进程已经在关闭、撤销这个插口了。如果不采取有效的措施加以防范，就可能出现严重问题：当 client 方进程在自以为找到了 server 插口的 sock 结构，并且通过 other 指针对其操作（例如将报文挂入它的 receive_queue 队列中）时，事实上 server 方进程已经释放了这个 sock 结构的空间。那么，内核中采取了什么样的措施来防止这种情况的发生呢？

(1) 在 sock 结构中设置了一个访问计数器 refcnt。每当某一进程卷入到这个结构的使用时，或者当所属的插口建立起一个连接时，就要通过 sock_hold()将这个计数器加 1。前面我们提到过，当 client 方进程从 unix_socket_table 表中的队列里找到 server 插口的 sock 数据结构时，就会通过 sock_hold()将这个结构中的访问计数 refcnt 加 1，其代码已经在前一节中看到过了。

(2) 相应地，每当一个进程结束了对一个插口的使用时，或者拆除一个连接时，都要通过另一个

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

inline 函数 sock_put()将相应 sock 结构中的计数器 refcnt 减 1。只有在这个计数器达到了 0 时，才允许（并且必须）将这个 sock 结构释放：

==================== include/net/sock.h 991 996 ====================

```
/* Ungrab socket and destroy it, if it was the last reference. */
static inline void sock_put(struct sock *sk)
{
	if (atomic_dec_and_test(&sk->refcnt))
		sk_free(sk);
}
```

==================== net/core/sock.c 585 606 ====================

[sock_put()>sk_free()]

```
585 void sk_free(struct sock *sk)
586 {
587 #ifdef CONFIG_FILTER
588 	struct sk_filter *filter;
589 #endif
590 
591 	if (sk->destruct)
592 		sk->destruct(sk);
593 
594 #ifdef CONFIG_FILTER
595 	filter = sk->filter;
596 	if (filter) {
597 		sk_filter_release(sk, filter);
598 		sk->filter = NULL;
599 	}
600 #endif
601 
602 	if (atomic_read(&sk->omem_alloc))
603 		printk(KERN_DEBUG "sk_free: optmem leakage (%d bytes) detected.\n",
atomic_read(&sk->omem_alloc));
604 
605 	kmem_cache_free(sk_cachep, sk);
606 }
```

注意 592 行对 destruct 函数的调用，这个函数指针是在分配 sock 数据结构时设置好了的，用来在释放缓冲区之前完成一些必要的附加操作。605 行的 kmem_cache_free()则将给定的 sock 结构在其 slab 中加以释放。

(3) 关闭、撤销一个插口时，其 sock 结构中的计数 refcnt 有可能还大于 1，表示还有用户，所以不能马上将这个结构释放，而只能将其 refcnt 计数减 1，把释放该结构的责任留给最后将这个计数减到 0 的那个进程。但是，光凭计数 refcnt 不足以说明该插口在逻辑上是否已经撤销，所以在 sock 结构中又设置了一个标志量 dead，表示尽管该 sock 结构中的 refcnt 还不是 0，所以还不能把数据结构最后释放，但实际上插口已经不存在了。

(4) 光是对 sock 结构的使用和释放加以保护还不够，还要防止对 sock 结构的使用（例如报文的到

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

达）和撤销在时间上相重叠。也就是说，这二者在时间上必须加以“串行化”。这样，如果插口的撤销在前，那就让撤销的过程先完成，而对 sock 的使用则就此打住。反之，如果对 sock 结构的使用在前，那就让使用的过程先完成，然后再来撤销，因为在撤销的过程中可能需要对使用的后果（例如链入到 receive_queue 队列中的报文）加以善后处理。为此目的，内核中设置了两对加锁/解锁操作，即 unix_state_rlock()/unix_state_runlock() 和 unix_state_wlock()/unix_state_wunlock()。当一个进程要读取 sock 结构中的状态信息（特别是 dead）时，要先调用 unix_state_rlock()加锁。这样，如果另一个进程正想要改变 sock 结构中的状态信息（例如想要把 dead 变成 1），就要在一个循环中（并不睡眠！）等待解锁后才能继续。反过来，如果一个进程要改变 sock 结构中的状态信息，则要先通过 unix_state_wlock()加锁。这样，想要读的进程就会在一个循环中等待解锁。这里 unix_state_rlock() 和 unix_state_wlock()是互锁的，如果 unix_state_rlock()先成功了，那么调用 unix_state_wlock()的进程就会在这个函数中循环等待（所以叫 spinlock），反之亦然。

以上这些措施防止了前述问题的发生。在 unix_stream_connect()的代码中，读者可以看到在检查 other->dead 之前先调用了 unix_state_rlock()。如果此时 server 方进程正在撤销该插口（及其 sock 数据结构）的过程中，并且赶在前面调用了 unix_state_wlock()，则当前进程就会在 unix_state_rlock()中循环等待，直至撤销 sock 数据结构的操作完成。但是此时由于 sock 结构中的 refcnt 计数至少为 2（因为当前进程在从杂凑表 unix_socket_table 的队列中找到这个结构时已经递增了这个计数），所以 server 方进程不会把 sock 结构释放。如果当前进程检测到 other->dead 非 0，就知道这个插口实际上已经不存在了。所以一方面把锁打开，一方面调用 sock_put()将其 refcnt 计数减 1，如果减 1 以后 refcnt 变成了 0，就担负起释放这个数据结构的责任。然后，就转回标号 restart 处，看看杂凑表的队列中是否还有相同地址的其它 sock 数据结构。当然，通常会因为找不到而使本次连接请求夭折（见 897 行）。

过了这一关，进而要检查目标插口的状态。只有处于 TCP_LISTEN 状态的插口才允许接受连接请求。注意，这与 server 方进程是否已经调用了 accept()或者正在 accept()中睡眠等待连接请求无关。

在 server 插口的 sock 结构中已经准备好了容纳连接请求报文的队列。不过，这个队列的长度是有限制的，在 sys_listen()中已经设置好了这个队列的最大长度。当队列长度超过这个最大长度时，就再不能把新的连接请求链入到队列中了。此时 client 方进程根据预定的等待时间决定是立即出错返回或是睡眠等待。如果决定睡眠等待，就在 918 行调用 unix_wait_for_peer()进入睡眠（net/unix/af_unix.c）：

==================== net/unix/af_unix.c 830 850 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>unix_wait_for_peer()]

```
830  static long unix_wait_for_peer(unix_socket *other, long timeo)
831  {
832       int sched;
833       DECLARE_WAITQUEUE(wait, current);
834
835       __set_current_state(TASK_INTERRUPTIBLE);
836       add_wait_queue_exclusive(&other->protinfo.af_unix.peer_wait, &wait);
837
838       sched = (!other->dead &&
839            !(other->shutdown&RCV_SHUTDOWN) &&
840            skb_queue_len(&other->receive_queue) > other->max_ack_backlog);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
841
842         unix_state_runlock(other);
843
844         if (sched)
845              timeo = schedule_timeout(timeo);
846
847         __set_current_state(TASK_RUNNING);
848         remove_wait_queue(&other->protinfo.af_unix.peer_wait, &wait);
849         return timeo;
850   }
```

与前一节中 server 方在 wait_for_packet()中睡眠等待相似，这里也是在当前进程的系统空间堆栈上分配一个 wait_queue_t 数据结构，将其挂入目标插口的 peer_wait 队列，然后就通过 schedule_timeout()进入定时的睡眠，醒来时再从该队列中脱链。在 unix_accept()的代码中我们看到，当 server 方进程从队列中接收了一个连接请求，从而使队列长度有所下降时，就要唤醒一个（如果有的话）正等待着要将连接请求挂入该队列的进程。当进程被唤醒而从 unix_wait_for_peer()中返回时，要通过 sock_put()释放目标插口，再检查是否因为接收到了信号而被唤醒。如果是就出错返回，提前结束本次系统调用；否则就转回到标号 restart 处，从杂凑表中的队列重新开始（见 924 行），因为情况可能已经改变了。

为什么这里与前一节中不同，在接收到了信号而被唤醒时要提前结束本次系统调用呢？我们不妨看看这里等待的是什么。如前所述，这里等待的是 server 方的报文队列中出现空间而使报文得以投递，可是那并不意味着操作的完成，本次系统调用要到连接请求被接受时才会完成。显然，那并不是在一个有限的、可以预测的短时间内能够完成的，所以只好使系统调用提前结束，先来处理已经到达的信号。

过了这一关，目标插口，即 server 插口一方已经没有问题了，但是 client 插口自己这一方呢？我们继续往下看函数 unix_stream_connect()的代码（net/unix/af_unix.c）。

===================== net/unix/af_unix.c 927 961 =====================

[sys_socketcall()>sys_connect()>unix_stream_connect()]

```
927         /* Latch our state.
928
929            It is tricky place. We need to grab write lock and cannot
930            drop lock on peer. It is dangerous because deadlock is
931            possible. Connect to self case and simultaneous
932            attempt to connect are eliminated by checking socket
933            state. other is TCP_LISTEN, if sk is TCP_LISTEN we
934            check this before attempt to grab lock.
935
936            Well, and we have to recheck the state after socket locked.
937          */
938         st = sk->state;
939
940         switch (st) {
941         case TCP_CLOSE:
942              /* This is ok... continue with connect */
943              break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
944     case TCP_ESTABLISHED:
945          /* Socket is already connected */
946          err = -EISCONN;
947          goto out_unlock;
948     default:
949          err = -EINVAL;
950          goto out_unlock;
951     }
952
953     unix_state_wlock(sk);
954
955     if (sk->state != st) {
956          unix_state_wunlock(sk);
957          unix_state_runlock(other);
958          sock_put(other);
959          goto restart;
960     }
961
```

代码中 940 行的 switch 语句对 client 方插口的状态进行检查，看看是否处于 TCP_CLOSE 状态。读者也许要问，这个检查为什么不放在 sys_connect()或者 unix_stream_connect()一开头的地方呢？那样如果状态不对的话。开始就可以回头了，岂不是可以省去这么多麻烦？答案是，即使在一开始时就作了这种检查，现在也还得再检查，因为情况可能中途有改变。要知道，可能会有多个进程（创建了插口的进程可能会 fork()出一批子进程）并发地对同一个插口发动操作，从而让另一个进程抢了先。另一方面，一个 client 方的进程也不能一开始就把 sock 结构锁住不放，因为在 unix_stream_connect()中可能会进入睡眠（否则就有可能导致死锁了，请读者想想为什么？）所以，不管怎样，在这个地方总是需要对 client 插口的状态进行检查的。通过了检查以后，就要改变其状态了，所以调用 unix_state_wlock()将其锁住（见 952 行）。可是在加锁成功以后还得再检查一次（见 955 行）！为什么呢？须知 unix_state_wlock()中可能隐藏着一个循环等待，而引起循环等待的原因是另一个进程（在另一个 CPU 上运行）抢先把它锁住了，所以在循环等待的前后这个 sock 结构的状态可能就不一样了。这一段代码（938～960 行，直到结束）看似平常，实际上却极有讲究，对于进程间的同步与互斥是一段很好的教材，建议读者把它读透，并且多问几个为什么，再自己来解答。

接着，就要来设置有关的数据结构了（net/unix/af_unix.c）：

==================== net/unix/af_unix.c 962 993 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()]

```
962     /* The way is open! Fastly set all the necessary fields... */
963
964     sock_hold(sk);
965     unix_peer(newsk)=sk;
966     newsk->state=TCP_ESTABLISHED;
967     newsk->type=SOCK_STREAM;
968     newsk->peercred.pid = current->pid;
969     newsk->peercred.uid = current->euid;
970     newsk->peercred.gid = current->egid;
```

```
971       newsk->sleep = &newsk->protinfo.af_unix.peer_wait;
972
973       /* copy address information from listening to new sock*/
974       if (other->protinfo.af_unix.addr)
975       {
976            atomic_inc(&other->protinfo.af_unix.addr->refcnt);
977            newsk->protinfo.af_unix.addr=other->protinfo.af_unix.addr;
978       }
979       if (other->protinfo.af_unix.dentry) {
980            newsk->protinfo.af_unix.dentry=dget(other->protinfo.af_unix.dentry);
981            newsk->protinfo.af_unix.mnt=mntget(other->protinfo.af_unix.mnt);
982       }
983
984       /* Set credentials */
985       sk->peercred = other->peercred;
986
987       sock_hold(newsk);
988       unix_peer(sk)=newsk;
989       sock->state=SS_CONNECTED;
990       sk->state=TCP_ESTABLISHED;
991
992       unix_state_wunlock(sk);
993
```

这里涉及的数据结构有这么几个：指针 sock 指向 client 插口的 socket 结构，指针 sk 指向 client 插口的 sock 结构，指针 other 指向 server 插口的 socket 结构，而指针 newsk 则指向一个新的 sock 结构。这个新的 sock 结构准备传给 server 方进程，以供在 accept()中新创建的插口配对使用。这里的 unix_peer()是个宏操作，其定义在文件 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 140 140 ====================

```
140  #define unix_peer(sk) ((sk)->pair)
```

所以，经过这段程序以后，newsk->pair 指向 sk，而 sk->pair 指向 newsk（这就是 Unix 域中插口间连接的主体)。现在二者已经挂上了钩，建立起连接，所缺少的环节就是让 newsk 跟 server 方进程在 accept()中新创建的插口挂上钩了。另一方面，client 方插口的 sock 结构的状态也变成了 TCP_ESTABLISHED。这样，如果有另一个进程也要在这个插口上进行 connect()操作，就会在上面的 944 行受到阻拦而失败返回。

代码中 980 行的 dget()实际上只是递增 dentry 结构中的共享计数 d_count，我们已经在“文件系统”一章中看到过了。981 行的 mntget()也类似。

我们尚未将连接请求报文，即 sk_buff 结构挂入 server 方 sock 结构的 receive_queue 队列中（代码中的指针 skb 指向这个结构)，下面就要来做这件事了。继续在 net/unix/af_unix.c 中往下看：

==================== net/unix/af_unix.c 994 1013 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()]

```
994       /* take ten and and send info to listening sock */
995       skb_queue_tail(&other->receive_queue,skb);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    unix_state_runlock(other);
    other->data_ready(other, 0);
    sock_put(other);
    return 0;

out_unlock:
    if (other)
        unix_state_runlock(other);

out:
    if (skb)
        kfree_skb(skb);
    if (newsk)
        unix_release_sock(newsk, 0);
    if (other)
        sock_put(other);
    return err;
}
```

这里 skb_queue_tail()是个 inline 函数，其代码在 include/linux/skbuff.h 中：

==================== include/linux/skbuff.h 463 482 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>skb_queue_tail()]

```
/**
 *  skb_queue_tail - queue a buffer at the list tail
 *  @list: list to use
 *  @newsk: buffer to queue
 *
 *  Queue a buffer at the tail of the list. This function takes the
 *  list lock and can be used safely with other locking &sk_buff functions
 *  safely.
 *
 *  A buffer cannot be placed on two lists at the same time.
 */

static inline void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
{
    unsigned long flags;

    spin_lock_irqsave(&list->lock, flags);
    __skb_queue_tail(list, newsk);
    spin_unlock_irqrestore(&list->lock, flags);
}
```

==================== include/linux/skbuff.h 449 461 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>skb_queue_tail()>__skb_queue_tail()]

```
static inline void __skb_queue_tail(struct sk_buff_head *list, struct sk_buff *newsk)
{
    struct sk_buff *prev, *next;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
452
453       newsk->list = list;
454       list->qlen++;
455       next = (struct sk_buff *)list;
456       prev = next->prev;
457       newsk->next = next;
458       newsk->prev = prev;
459       next->prev = newsk;
460       prev->next = newsk;
461  }
```

最后，在 sock 结构中有个函数指针 data_ready，每当将一个报文（无论控制报文或是数据报文）链入到一个 sock 结构的 receive_queue 队列中以后，都要通过这个函数指针来调用一个预先设置好的函数（997 行），这种情况常称为 " call back"。在创建插口时所调用的函数 sock_init_data()里，函数指针 data_ready 被设置成指向 sock_def_readable()，所以这里的 " call back" 就是对这个函数的调用，其代码在 net/core/sock.c 中：

==================== net/core/sock.c 1083 1090 ====================
[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_def_readable()]

```
1083 void sock_def_readable(struct sock *sk, int len)
1084 {
1085      read_lock(&sk->callback_lock);
1086      if (sk->sleep && waitqueue_active(sk->sleep))
1087           wake_up_interruptible(sk->sleep);
1088      sk_wake_async(sk,1,POLL_IN);
1089      read_unlock(&sk->callback_lock);
1090 }
```

对于此时要做的事情，读者大概至少已经猜对了一半，那就是要唤醒可能正在睡眠中等待着连接请求的 server 方进程（1094 行）。但是，另一半，也就是 sock_wake_async()，是干什么用的呢？让我们回顾一下 server 方进程是怎样通过 accept()来接受连接请求的。大家知道，accept()是一个 server 插口接受连接请求的惟一途径，server 插口是不能主动要求连接的。同时，accept()的操作从本质上说是"同步"的，如果调用 accept()时尚无连接请求到来，就要睡眠等待。诚然，server 方的进程可以通过 O_NONBLOCK 标志让 accept()在没有连接请求时立即返回，但这样一来，server 方进程就只好循环地或者定期地调用 accept()来测试是否有连接请求到来。再考虑有时候 server 方进程要同时照看好几个 server 插口的情况，这时候 server 方进程就只好将 O_NONBLOCK 标志设成 1 来"轮询"各个 server 插口了，但是，不管是睡眠也好，或是轮询也好，server 进程在此期间就不能做别的事情了。所以，O_NONBLOCK 标志并不改变 accept()的同步本质。

那么，有没有办法"异步"地等待连接呢？就是说使 server 方进程可以干点别的，到有连接请求到来时就通知它，让它到那时候再来调用 accept()。答案是肯定的。我们以前讲过，硬件（包括处理器）层次上的异步通信手段是中断，而软件层次上的异步通信手段是"信号"。显然，我们在这里也可以利用信号机制。简而言之，就是让内核在有连接请求到来时就向 server 方进程发一个信号，而 server 方进程则平时可以处理别的事情，只是在接收到有关信号时就来调用 accept()。事实上，并不是只有插口的连接才会有这样的要求，异步操作已经成为文件系统操作的一个组成部分。在文件操作 ioctl()中就设

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

置了一条命令 FIOASYNC，让有关进程（必须是文件的主人）可以通过这条命令向一个文件挂上号，让它在某种条件得到满足时就向该进程发送一个信号。为了这个目的，在插口的 socket 结构中设置了一个队列 fasync_list。当 server 方进程希望异步地等待连接时，就通过 ioctl()的 FIOASYNC 命令（插口也是一个已打开文件）将一个 fasync_struct 结构挂入到这个队列中。另一方面，我们以前也讲过，代表着一个插口的 file 结构中有个指针 f_op，指向一个 file_operations 结构 socket_file_ops。这个结构中的指针 fasync 指向函数 sock_fasync()，当 server 方进程通过 ioctl()发出 FIOASYNC 命令时，就会执行这个函数来完成上述将一个 fasync_struct 数据结构挂入到这个队列中的操作。

所以，在 " call back" 函数 sock_def_readable()中的另一件事就是：通过 sk_wake_async()给可能已经在队列 fasync_list 中挂上号的进程发信号，其代码在 include/net/sock.h 中：

==================== include/net/sock.h 1219 1223 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_def_readable()>sk_wake_async()]

```
static inline void sk_wake_async(struct sock *sk, int how, int band)
{
	if (sk->socket && sk->socket->fasync_list)
		sock_wake_async(sk->socket, how, band);
}
```

==================== net/socket.c 786 811 ====================

[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_def_readable()>sk_wake_async()>sock_wake_async()]

```
/* This function may be called only under socket lock or callback_lock */

int sock_wake_async(struct socket *sock, int how, int band)
{
	if (!sock || !sock->fasync_list)
		return -1;
	switch (how)
	{
	case 1:

		if (test_bit(SOCK_ASYNC_WAITDATA, &sock->flags))
			break;
		goto call_kill;
	case 2:
		if (!test_and_clear_bit(SOCK_ASYNC_NOSPACE, &sock->flags))
			break;
		/* fall through */
	case 0:
	call_kill:
		__kill_fasync(sock->fasync_list, SIGIO, band);
		break;
	case 3:
		__kill_fasync(sock->fasync_list, SIGURG, band);
	}
	return 0;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里调用时的参数 how 为 1，band 为 POLL_IN。我们把这段代码留给读者自己阅读。其中函数__kill_fasync()的代码在 fs/fcntl.c 中，它扫描整个 fasync_list 队列，向每个挂上号的进程发出信号。

==================== fs/fcntl.c 482 499 ====================
[sys_socketcall()>sys_connect()>unix_stream_connect()>sock_def_readable()>sk_wake_async()>sock_wake_async()>__kill_fasync()]

```
void __kill_fasync(struct fasync_struct *fa, int sig, int band)
{
	while (fa) {
		struct fown_struct * fown;
		if (fa->magic != FASYNC_MAGIC) {
			printk(KERN_ERR "kill_fasync: bad magic number in "
			       "fasync_struct!\n");
			return;
		}
		fown = &fa->fa_file->f_owner;
		/* Don't send SIGURG to processes which have not set a
		   queued signum: SIGURG has its own default signalling
		   mechanism. */
		if (fown->pid && !(sig == SIGURG && fown->signum == 0))
			send_sigio(fown, fa->fa_fd, band);
		fa = fa->fa_next;
	}
}
```

不言而喻，如果要异步地接受连接请求，则 server 方的进程必须事先设置好相应的信号处理程序。

至此，sys_connect()已经完成了它的任务（除 sockfd_put()以外）。如果 server 方进程已经在 accept()中等待，则唤醒以后就会来补上缺失的一环，就是使新创建的 socket 结构与通过 sk_buff 结构传过来的 sock 结构挂上钩（见 unix_accept()）代码中的 1068 行，注意那里的指针 newsock 指向由 accept()新创建的 socket 结构，而指针 tsk 指向由 sys_connect()新创建的 sock 结构）。这样，就完成了两个 unix 域插口之间的连接。如果 server 方进程尚未调用 accept()，则当它调用 accept()时连接请求已经在 server 插口的 receive_queue 队列中等待，所以无需睡眠等待就可以完成这种连接。

再来看"无连接"模式的 connect()操作。对于 Unix 域"无连接"模式的插口，所用的 proto_ops 数据结构为 unix_dgram_ops，而相应的函数指针 connect 则指向 unix_dgram_connect()。

以前讲过，既然是"无连接"模式的插口，本来就没有"建立连接"这一说。之所以也有个 connect()操作，只是要利用它来将本来每次发送报文时都要重复的一些操作集中在一起，以避免浪费。有些什么"每次发送报文时都要重复的操作"呢？

(1) 每次都要从用户空间把对方的插口地址拷贝到系统空间中。

(2) 在网络环境下，通常需要根据对方的地址从路径表中查得应该使用的网络接口，以及可能需要的在网络层和 / 或链路层上使用的地址。并且常常还要进行从符号地址到网络地址数值的转换。

(3) 在 Unix 域中，则通常需要根据对方的插口地址从文件系统中打开相应的文件（节点），然后用索引节点的号码在杂凑表的某个队列中找到对方的 sock 数据结构。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

就每一次报文发送来说，这些开销不能算大。但是，如果发送成千上万个报文到同一个目标插口，这总共的开销就不能小看了。下面，我们就来看看，sys_connect()对于 Unix 域“无连接”模式的插口到底做了些什么。函数 unxi_dgram_connect()的代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 770 828 ====================

[sys_socketcall()>sys_connect()>unix_dgram_connect()]

```
static int unix_dgram_connect(struct socket *sock, struct sockaddr *addr,
                   int alen, int flags)
{
    struct sock *sk = sock->sk;
    struct sockaddr_un *sunaddr=(struct sockaddr_un*)addr;
    struct sock *other;
    unsigned hash;
    int err;

    if (addr->sa_family != AF_UNSPEC) {
        err = unix_mkname(sunaddr, alen, &hash);
        if (err < 0)
            goto out;
        alen = err;

        if (sock->passcred && !sk->protinfo.af_unix.addr &&
           (err = unix_autobind(sock)) != 0)
            goto out;

        other=unix_find_other(sunaddr, alen, sock->type, hash, &err);
        if (!other)
            goto out;

        unix_state_wlock(sk);

        err = -EPERM;
        if (!unix_may_send(sk, other))
            goto out_unlock;
    } else {
        /*
         *   1003.1g breaking connected state with AF_UNSPEC
         */
        other = NULL;
        unix_state_wlock(sk);
    }

    /*
     * If it was connected, reconnect.
     */
    if (unix_peer(sk)) {
        struct sock *old_peer = unix_peer(sk);
        unix_peer(sk)=other;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
812             unix_state_wunlock(sk);
813
814             if (other != old_peer)
815                 unix_dgram_disconnected(sk, old_peer);
816             sock_put(old_peer);
817         } else {
818             unix_peer(sk)=other;
819             unix_state_wunlock(sk);
820         }
821         return 0;
822
823 out_unlock:
824         unix_state_wunlock(sk);
825         sock_put(other);
826 out:
827         return err;
828 }
```

读者已经学习过“有连接”模式插口的 accept()和 connect()操作的代码，对于这里所调用或引用的大部分函数和宏定义都已经熟悉了。事实上，读者在前几节中尚未见到过的函数只有一个，那就是 unix_may_send()，其代码也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 140 150 ====================

[sys_socketcall()>sys_connect()>unix_dgram_connect()>unix_may_send()]

```
#define unix_peer(sk) ((sk)->pair)

extern __inline__ int unix_our_peer(unix_socket *sk, unix_socket *osk)
{
    return unix_peer(osk) == sk;
}

extern __inline__ int unix_may_send(unix_socket *sk, unix_socket *osk)
{
    return (unix_peer(osk) == NULL || unix_our_peer(sk, osk));
}
```

也就是说，unix_may_send()检查对方插口的 sock 结构中的指针 pair，看看它是否已经指向某个 sock 结构，如果是，就进而检查它所指向的是否正是我方插口的 sock 结构。如果对方 sock 结构中的指针 pair 已经指向了另一个插口的 sock 结构，那就只好就此打住了。这种情况下，unix_dgram_conncct()返回出错代码-EPERM，表示不允许。也就是说，对方插口必须尚未连接到别的插口，改变对方插口与其它插口的连接是不允许的。可是，如果我方插口原来已经有了“连接”，则允许以新的连接代替原有的连接（见 809～816 行），而不需要特别将原有的连接拆除。这里所谓“连接”只不过是让我方 sock 结构中的指针 pair 指向对方的 sock 结构。注意，这个“连接”只是单向的，对方插口的数据结构丝毫不受影响，对方进程有完全的自由使其插口指向（或曰“连接到”）其它插口。此外，“无连接”插口也不像“有连接”插口那样有个“状态机”，所以“无连接”通信有时也称作“无状态”通信；对方插口也不会像“有连接”模式的 server 插口那样生下一个“蛋”来。正因为这样，unix_dgram_connect()的

代码比 unix_stream_connect()的代码要简单多了。

7.7 报文的接收与发送

插口上的报文接收与发送有两个程序设计界面。第一个界面是为插口专设的，从用户程序的角度来看就是三对 libc 库函数，即 recv()/send()、recvfrom()/sendto()以及 recvmsg()/sendmsg()，但是最终都归结于一个统一的系统调用，在内核中的入口为 sys_socketcall()。不过，除特殊的应用外（后面会讲到），这些库函数并不是非得成对地使用。也就是说，原则上双方都可以自由地选择使用三者之一，但是 send()只能在已经通过 connect()建立了连接，或确定了目标插口以后才可使用。另一方面，从语义的角度来说，只要已经使用了 connect()就应该用 send()而不是 sendto()，因为既然目标已经确定了，就不应该再在发送时规定目标了。不过，尽管如此，用 sendto()也还是可以的，只不过要把调用参数中的对方地址（指针）设成 NULL，把地址长度设成 0。读者和前面已经看到过，在内核中 sys_recv()实际上就是通过 sys_recvfrom()实现的，而 sys_send()则是通过 sys_sendto()实现的，无非就是把地址指针设成 NULL，把地址长度设成 0 而已。此外，下面读者还会看到，实际上三个用于接收的函数最后全都通过 sock_recvmsg()来接收报文，而三个用于发送的函数则全都是通过 sock_sendmsg()来发送报文的。

第二个界面是通过常规的文件操作 read()和 write()这两对系统调用来进行的（还有 readv()和 writev()，与 recvmsg()和 sendmsg()相似）。读者不妨回过去看看前面的联系图（图 7.1）。从当前进程的 task_struct 结构开始，通过插口的打开文件号找到相应的 file 结构；再顺着 file 结构中的指针 f_op，就可以找到这个“文件”的 file_operations 结构 socket_file_ops。这就是文件操作的跳转表，是在 socket.c 中定义的：

==================== net/socket.c 114 126 ====================

```
static struct file_operations socket_file_ops = {
	llseek:		sock_lseek,
	read:		sock_read,
	write:		sock_write,
	poll:		sock_poll,
	ioctl:		sock_ioctl,
	mmap:		sock_mmap,
	open:		sock_no_open,	/* special open code to disallow open via /proc */
	release:	sock_close,
	fasync:		sock_fasync,
	readv:		sock_readv,
	writev:		sock_writev
};
```

由此可见，相应的函数为 sock_read()和 sock_write()，这两个函数的代码都在 net/socket.c 中。这里要提醒一下，插口一经创建就已经打开了（返回打开文件号），并没有一般文件那样的 open()操作，所以指针 open 指向一个函数 sock_no_open()。

函数 sock_write()与 sys_sendto()很相似，我们不妨比较一下 sock_write()和 sys_sendto()的代码。先看 sock_write()：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== net/socket.c 571 603 ====================

[sys_write()>sock_write()]

```
/*
 *   Write data to a socket. We verify that the user area ubuf..ubuf+size-1
 *   is readable by the user process.
 */

static ssize_t sock_write(struct file *file, const char *ubuf,
                size_t size, loff_t *ppos)
{
     struct socket *sock;
     struct msghdr msg;
     struct iovec iov;

     if (ppos != &file->f_pos)
          return -ESPIPE;
     if(size==0)        /* Match SYS5 behaviour */
          return 0;

     sock = socki_lookup(file->f_dentry->d_inode);

     msg.msg_name=NULL;
     msg.msg_namelen=0;
     msg.msg_iov=&iov;
     msg.msg_iovlen=1;
     msg.msg_control=NULL;
     msg.msg_controllen=0;
     msg.msg_flags=!(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
     if (sock->type == SOCK_SEQPACKET)
          msg.msg_flags |= MSG_EOR;
     iov.iov_base=(void *)ubuf;
     iov.iov_len=size;

     return sock_sendmsg(sock, &msg, size);
}
```

再看 sys_sendto()：

==================== net/socket.c 1159 1201 ====================

[sys_socketcall()>sys_sendto()]

```
/*
 *   Send a datagram to a given address. We move the address into kernel
 *   space and check the user space data area is readable before invoking
 *   the protocol.
 */

asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags,
                 struct sockaddr *addr, int addr_len)
{
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
    struct socket *sock;
    char address[MAX_SOCK_ADDR];
    int err;
    struct msghdr msg;
    struct iovec iov;

    sock = sockfd_lookup(fd, &err);
    if (!sock)
        goto out;
    iov.iov_base=buff;
    iov.iov_len=len;
    msg.msg_name=NULL;
    msg.msg_iov=&iov;
    msg.msg_iovlen=1;
    msg.msg_control=NULL;
    msg.msg_controllen=0;
    msg.msg_namelen=addr_len;
    if(addr)
    {
        err = move_addr_to_kernel(addr, addr_len, address);
        if (err < 0)
            goto out_put;
        msg.msg_name=address;
    }
    if (sock->file->f_flags & O_NONBLOCK)
        flags |= MSG_DONTWAIT;
    msg.msg_flags = flags;
    err = sock_sendmsg(sock, &msg, len);

out_put:
    sockfd_put(sock);
out:
    return err;
}
```

可见二者几乎是一样的，最后都是调用 sock_sendmsg()来完成任务。同样的相似性也存在于 sock_read()和 sys_recvmsg()之间，并且二者都通过 sock_recvmsg()来完成任务。也就是说，两个界面上的这些函数最后都是殊途同归，都归结到 sock_recvmsg()和 sock_sendmsg()两个函数。所以，两个界面其实没有多大区别。不过，从语义的角度来说，一般对“有连接”插口倾向于使用 read()/write()，而对“无连接”插口则通常都使用 recvfrom()/sendto()等函数。这是因为在“有连接”模式的通信中将传递的数据看成连续的“字节流”，而不保留“报文”的边界（所以其类型称为 SOCK_STREAM），与文件操作的语义比较贴近。反之，“无连接”模式的通信则是“面向报文”的，所以保留报文的边界。

我们以接收方为例进一步说明这两种模式间的区别。假定在一个“无连接”插口的 receive_queue 队列中已经有两个报文在等待读取，每个报文都含有 200 字节的数据。然后，接收方进程通过 recv()来接收，缓冲区的大小为 150 字节。由于缓冲区小于队列中第一个报文的大小，所以只能接收 150 个字节，于是 recv()返回 150，即实际接收到的字节数。此时虽然第一个报文中尚有 50 字节的剩余，但

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

是内核却会将其从队列中脱链并释放，剩余的 50 字节就丢弃了。这是因为“无连接”模式的通信是以报文为单位，而不是以实际报文中的字节为单位的。反之，如果缓冲区的大小是 300 字节，那么虽然缓冲区大于第一个报文的实际大小，却不会再到第二个报文中去读取一部分，以填满缓冲区，所以 recv()返回 200。可是，如果插口是“有连接”模式的，那就不同了。在第一种情况中，剩余的 50 字节会保留下来供下一次继续读取，而在第二种情况下则会在读取了第一个报文中的 200 字节之后再到第二个报文中读取 100 字节，而将第二个报文中剩余的 100 字节留待下一次继续读取，所以此时 recv()返回 300。当然，“有连接”和“无连接”两种通信模式的区别远不止于此。但是，对 Unix 域的插口来说，可靠性的问题实际上并不存在（因为不涉及网络介质），保序性的问题也不存在（因为不涉及不同的路径），剩下的就是连接的建立以及语义上的这种区别了。从用户程序设计的角度来看，后者实际上更为重要。举例来说，如果用户程序未经建立连接就试图通过一个“有连接”插口发送信息，那么 send()马上就会出错返回，从而一开始就可以发现这个问题。然而，如果是在“无连接”插口上用 read()来接收，并且在程序设计中误以为接收的内容是连续的字节流，则系统调用本身并不会出错返回，可是却会出现一些似乎是莫明其妙的毛病而又不容易找到原因。反过来，将互相独立的报文作为连续的字节流来接收也会造成问题。

在深入到 sock_recvmsg()和 sock_sendmsg()的代码中去之前，还要先回到前面 sys_sendto()的代码中去看一下这两个函数的外围。由于这两个函数要适应上述所有函数的需要，所以是按其中最复杂的 sys_sendmsg()和 sys_recvmsg()的要求而设计的。因此，在 sys_sendto()中要先使用传递下来的参数，组装起一个“报头”，即 msghdr 结构，用作调用 sock_sendmsg()的主要参数。有关定义见 include/linux/socket.h：

==================== include/linux/socket.h 27 53 ====================

```
/*
 *	As we do 4.4BSD message passing we use a 4.4BSD message passing
 *	system, not 4.3. Thus msg_accrights(len) are now missing. They
 *	belong in an obscure libc emulation or the bin.
 */

struct msghdr {
	void *	msg_name;	/* Socket name			*/
	int		msg_namelen;	/* Length of name		*/
	struct iovec *	msg_iov;	/* Data blocks			*/
	__kernel_size_t	msg_iovlen;	/* Number of blocks		*/
	void 	*	msg_control;	/* Per protocol magic (eg BSD file descriptor passing) */
	__kernel_size_t	msg_controllen;	/* Length of cmsg list */
	unsigned	msg_flags;
};

/*
 *	POSIX 1003.1g - ancillary data object information
 *	Ancillary data consits of a sequence of pairs of
 *	(cmsghdr, cmsg_data[])
 */

struct cmsghdr {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        __kernel_size_t    cmsg_len;/* data byte count, including hdr */
          int          cmsg_level;   /* originating protocol */
          int          cmsg_type;    /* protocol-specific type */
};
```

这里 msghdr 结构中的 msg_name、msg_namelen 以及 msg_flags 分别对应于 sys_sendto()的参数 addr、addr_len 以及 flags。指针 msg_control 可以指向一个附加的数据结构，用来提供一些附加的控制信息，其类型取决于具体的规程，一般为 cmsghdr 数据结构加 cmsg_data[]数组。此外，更重要的是，结构中有个指针 msg_iov 指向一个 iovec 结构数组，其定义在 include/linux/uio.h 中，而 msg_iovlen 则为该数组的大小：

==================== include/linux/uio.h 19 23 ====================

```
struct iovec
{
        void *iov_base;        /* BSD uses caddr_t (1003.1g requires void *) */
        __kernel_size_t iov_len; /* Must be size_t (1003.1g) */
};
```

数组中的每一个元素即 iovec 结构，都是一个所谓"io 向量"，由指向数据缓冲区的指针 iov_base 和表示缓冲区中数据长度的 iov_len 构成。这样，由报头 msghdr 结构所代表的报文可以由多个数据缓冲区构成，还可以包含由 msg_control 所指向的附加信息（通常是控制信息）。上述各个数据结构之间的联系如图 7.3 所示。

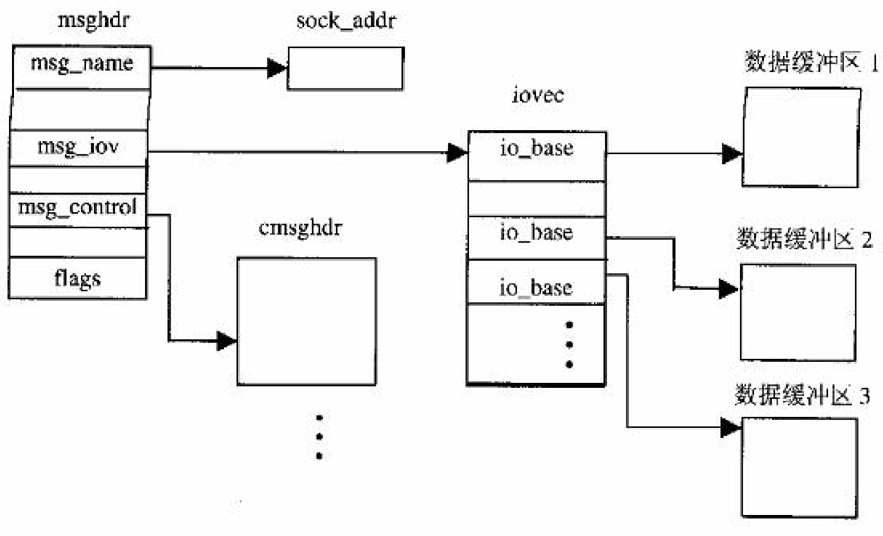


图 7.3　报文数据结构联系图

所以，msghdr 结构代表了报文的最为一般的形式。当报文中只有一个"io 向量"，即只有一个数据缓冲区，并且不含有控制信息时，就相当于普通的"缓冲区地址加数据长度"形式。那么，为什么在报文的一般形式中需要能容纳多个数据缓冲区呢？让我们考虑一下在网络环境下的报文接收，就以 Ethernet 为例吧。在 Ethernet 中，一个报文（packet，或 frame）的最大长度约为 1500 字节，而最小长

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

度只有数十字节。当开始从网络接收报文时，其长度通常是未知的，要到接收完毕时才能知道它的长度。另一方面，为了提高效率，用于从网络上接收报文的缓冲区都是预先分配好在一个“缓冲池”中备用的，缓冲池中所有的缓冲区都具有相同的长度。可是，每个缓冲区的大小以什么为准呢？如果都按可能的最大长度分配，那就必然造成很大的浪费，因为实际上在 Ethernet 中传递的报文多数是 100 字节以下的。如果减少缓冲区长度，那就势必有时候得用好几个缓冲区才能容纳一个报文。由此可见，比较合理的安排就是类似于 msghdr 数据结构那样的设计，选择适中的缓冲区长度，以多个缓冲区来容纳一个报文，允许在最后一个缓冲区中浪费少量的空间。不过要指出，这里讲的使用多个缓冲区是指在接收进程（或发送进程）与插口之间，而不是指插口与插口之间的报文缓冲区。

看了 sys_sendto()，如何将其参数组装成一个以 msghdr 结构为代表的报文的，则 sys_recvfrom()如何将其还原就是不言而喻的了。而我们在前面已经看到过 sys_send()和 sys_recv()分别是通过 sys_sendto()和 sys_recvfrom()实现的，只是将参数中的指针 addr 设成 NULL，整数 addrlen 设成 0 而已。至于 read()和 write()，我们已经看到其与 sock_write()和 sys_sendto()的相似性。

现在可以来看 sock_recvmsg()和 sys_recvfrom()的代码。先看前者（net/socket.c）：

==================== net/socket.c 514 525 ====================

[sys_socketcall()>sys_recvmsg()]

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
{
	struct scm_cookie scm;

	memset(&scm, 0, sizeof(scm));

	size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
	if (size >= 0)
		scm_recv(sock, msg, &scm, flags);

	return size;
}
```

对于 Unix 域，根据插口类型的不同，具体的函数可以是 unix_dgram_recvmsg()或者 unix_stream_recvmsg()。

再看 sys_recvfrom()：

==================== net/socket.c 1212 1252 ====================

[sys_socketcall()>sys_recvfrom()]

```
/*
 *	Receive a frame from the socket and optionally record the address of the
 *	sender. We verify the buffers are writable and if needed move the
 *	sender address from kernel to user space.
 */

asmlinkage long sys_recvfrom(int fd, void * ubuf, size_t size, unsigned flags,
			     struct sockaddr *addr, int *addr_len)
{
	struct socket *sock;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct iovec iov;
    struct msghdr msg;
    char address[MAX_SOCK_ADDR];
    int err,err2;

    sock = sockfd_lookup(fd, &err);
    if (!sock)
        goto out;

    msg.msg_control=NULL;
    msg.msg_controllen=0;
    msg.msg_iovlen=1;
    msg.msg_iov=&iov;
    iov.iov_len=size;
    iov.iov_base=ubuf;
    msg.msg_name=address;
    msg.msg_namelen=MAX_SOCK_ADDR;
    if (sock->file->f_flags & O_NONBLOCK)
        flags |= MSG_DONTWAIT;
    err=sock_recvmsg(sock, &msg, size, flags);

    if(err >= 0 && addr != NULL && msg.msg_namelen)
    {
        err2=move_addr_to_user(address, msg.msg_namelen, addr, addr_len);
        if(err2<0)
            err=err2;
    }
    sockfd_put(sock);
out:
    return err;
}
```

可见，实际上 sys_recvfrom()的主体就是 sock_recvmsg()，其代码在 net/socket.c 中：

==================== net/socket.c 514 525 ====================

[sys_socketcall()>sys_recvfrom()>sock_recvmsg()]

```
514 int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
515 {
516     struct scm_cookie scm;
517 
518     memset(&scm, 0, sizeof(scm));
519 
520     size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
521     if (size >= 0)
522         scm_recv(sock, msg, &scm, flags);
523 
524     return size;
525 }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这个函数的代码与上面的 sys_recvmsg()一模一样。简单地说，这个函数就做两件事。第一件是接收报文中的数据以及附加信息，第二件就是对附加信息的处理。

接收附加信息要用到一个数据结构，那就是 scm_cookie 结构，是在 include/net/scm.h 中定义的：

==================== include/net/scm.h 4 20 ====================

```
/* Well, we should have at least one descriptor open
 * to accept passed FDs 8)
 */
#define SCM_MAX_FD (OPEN_MAX-1)

struct scm_fp_list
{
	int		count;
	struct file	*fp[SCM_MAX_FD];
};

struct scm_cookie
{
	struct ucred		creds;		/* Skb credentials */
	struct scm_fp_list	*fp;		/* Passed files		*/
	unsigned long		seq;		/* Connection seqno	*/
};
```

从数据结构的定义可以看出，这里主要有两种信息。一种是对方进程的“身份”（credentials）。数据结构类型 ucred 是在 include/linux/socket.h 中定义的：

==================== include/linux/socket.h 125 129 ====================

```
struct ucred {
	__u32	pid;
	__u32	uid;
	__u32	gid;
};
```

另一种就是有关打开文件的信息。BSD（以及 Linux）的插口机制允许一个进程把它的若干个已打开文件的访问权传递给，或者说“授予”另一个进程。一个进程最多可以同时打开的文件个数取决于一个常数 OPEN_MAX，定义于 include/linux/limits.h 中：

==================== include/linux/limits.h 9 9 ====================

```
#define OPEN_MAX         256  /* # open files a process may have */
```

可是，有一个位置是必须保留的，那就是代表着插口本身的那个已打开文件。所以在 scm_fd_list 结构中的 file 结构指针数组 fp[]的大小为（OPEN_MAX-1）。

首先是数据部分的接收。对于 Unix 域的插口，sock->ops 视插口类型的不同而指向 unix_dgram_ops 或 unix_stream_ops，二者的 recvmsg 指针分别指向 unix_dgram_recvmsg()和 unix_stream_recvmsg()。

我们先来看比较简单的 unix_dgram_recvmsg()（net/unix/af_unix.c）：

==================== net/unix/af_unix.c 1398 1460 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()]

```
{
    msg->msg_namelen = sizeof(short);
    if (sk->protinfo.af_unix.addr) {
        msg->msg_namelen=sk->protinfo.af_unix.addr->len;
        memcpy(msg->msg_name,
               sk->protinfo.af_unix.addr->name,
               sk->protinfo.af_unix.addr->len);
    }
}

static int unix_dgram_recvmsg(struct socket *sock, struct msghdr *msg, int size,
                   int flags, struct scm_cookie *scm)
{
    struct sock *sk = sock->sk;
    int noblock = flags & MSG_DONTWAIT;
    struct sk_buff *skb;
    int err;

    err = -EOPNOTSUPP;
    if (flags&MSG_OOB)
        goto out;

    msg->msg_namelen = 0;

    skb = skb_recv_datagram(sk, flags, noblock, &err);
    if (!skb)
        goto out;

    wake_up_interruptible(&sk->protinfo.af_unix.peer_wait);

    if (msg->msg_name)
        unix_copy_addr(msg, skb->sk);

    if (size > skb->len)
        size = skb->len;
    else if (size < skb->len)
        msg->msg_flags |= MSG_TRUNC;

    err = skb_copy_datagram_iovec(skb, 0, msg->msg_iov, size);
    if (err)
        goto out_free;

    scm->creds = *UNIXCREDS(skb);

    if (!(flags & MSG_PEEK))
    {
        if (UNIXCB(skb).fp)
            unix_detach_fds(scm, skb);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1436     }
1437     else
1438     {
1439          /* It is questionable: on PEEK we could:
1440             - do not return fds - good, but too simple 8)
1441             - return fds, and do not return them on read (old strategy,
1442               apparently wrong)
1443             - clone fds (I choosed it for now, it is the most universal
1444               solution)
1445 
1446               POSIX 1003.1g does not actually define this clearly
1447               at all. POSIX 1003.1g doesn't define a lot of things
1448               clearly however!
1449 
1450          */
1451          if (UNIXCB(skb).fp)
1452               scm->fp = scm_fp_dup(UNIXCB(skb).fp);
1453     }
1454     err = size;
1455 
1456 out_free:
1457     skb_free_datagram(sk,skb);
1458 out:
1459     return err;
1460 }
```

参数 flags 中有个标志位 MSG_OOB，表示此次接收的目的是所谓“ out-of-band”报文。在“有连接”模式的通信中，普通的数据报文在发送时都编上序号，接收时就按次序接收。但是，有些报文是用于控制目的（例如载送着“ ^c”的报文）而需要优先传递的，此种“编外”报文就称为 OOB 报文。可是，在“无连接”模式中每个报文都是独立的，也无所谓次序，根本不存在 OOB 报文这么个概念，所以对“无连接”插口调用 sock_recvmsg()时 MSG_OOB 标志位不应该为 1（见 1407 行）。

函数 skb_recv_datagram()从插口的接收队列中摘取，或者等待着从该队列中摘取一个载运着报文的 sk_buff 结构，读者在前面讲述 sys_accept()的一节中已看到过其代码，此处就不重复了。在我们这个情景中，假定队列中已经有报文存在，所以当前进程不需要睡眠等待。从接收队列摘取一个报文以后，队列中就多出了一个报文的位置，所以要唤醒一个可能正在睡眠、等待着要投递报文的进程。如果 msghdr 结构中的指针 msg_name 不是 NULL，也就是说接收进程为对方（发送方）的插口地址准备好了一个缓冲区，就要将对方的插口地址拷贝到一个临时的缓冲区中（见这里的 1419 行和 sys_recvfrom()代码中的第 1245 行），为下一步将其拷贝到用户空间的缓冲区作好准备。函数 unix_copy_addr()的代码也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1387 1396 ====================

[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>unix_copy_addr()]

```
1387 static void unix_copy_addr(struct msghdr *msg, struct sock *sk)
1388 {
1389     msg->msg_namelen = sizeof(short);
1390     if (sk->protinfo.af_unix.addr) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1391            msg->msg_namelen=sk->protinfo.af_unix.addr->len;
1392            memcpy(msg->msg_name,
1393                  sk->protinfo.af_unix.addr->name,
1394                  sk->protinfo.af_unix.addr->len);
1395       }
1396 }
```

下面就是对接收长度的处理了（1421～1424 行）。如果缓冲区的大小，也就是要求接收的数据大小超过报文的实际长度，那就按报文的实际长度接收。反之，如果缓冲区小于报文的实际长度，那就按缓冲区的大小接收，而报文中剩余的数据就丢掉了，所以将 msg_flags 中的 MSG_TRUNC 标志设成 1，表示报文被截尾了。长度一经确定，就可以把数据从 sk_buff 结构中拷贝到由 msghdr 结构中的 iovec[] 向量表所指示的缓冲区中，函数 skb_copy_datagram_iovec()的代码在 net/core/datagram.c 中：

==================== net/core/datagram.c 200 209 ====================
[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>skb_copy_datagram_iovec()]

```
/*
 *   Copy a datagram to an iovec.
 *   Note: the iovec is modified during the copy.
 */

int skb_copy_datagram_iovec(struct sk_buff *skb, int offset, struct iovec *to,
                  int size)
{
     return memcpy_toiovec(to, skb->h.raw + offset, size);
}
```

函数 memcpy_toiovec()的代码则在 net/core/iovec.c 中：

==================== net/core/iovec.c 76 103 ====================
[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>skb_copy_datagram_iovec()>memcpy_toiovec()]

```
/*
 *   Copy kernel to iovec. Returns -EFAULT on error.
 *
 *   Note: this modifies the original iovec.
 */

int memcpy_toiovec(struct iovec *iov, unsigned char *kdata, int len)
{
     int err = -EFAULT;

     while(len>0)
     {
          if(iov->iov_len)
          {
               int copy = min(iov->iov_len, len);
               if (copy_to_user(iov->iov_base, kdata, copy))
                    goto out;
               kdata+=copy;
               len-=copy;
```

```
                iov->iov_len-=copy;
                iov->iov_base+=copy;
            }
            iov++;
        }
        err = 0;
out:
        return err;
}
```

对于 Unix 域的插口，一般都只使用一个缓冲区，但是当然也可通过 recvmsg()来把报文接收到若干个较小的缓冲区中。

除报文中的数据外，还要把 sk_buff 结构中载送的有关对方身份的信息也拷贝到前面准备下的 scm_cookie 结构中去。注意，函数 unix_dgram_recvmsg()中第 1430 行的赋值语句所做的是整个 ucred 结构的复制，对 UNIXCREDS 的有关定义在 include/net/af_unix.h 中：

==================== include/net/af_unix.h 28 35 ====================

```
struct unix_skb_parms
{
        struct ucred        creds;          /* Skb credentials */
        struct scm_fp_list *fp;       /* Passed files         */
};

#define UNIXCB(skb)     (*(struct unix_skb_parms*)&((skb)->cb))
#define UNIXCREDS(skb)  (&UNIXCB((skb)).creds)
```

在 sk_buff 结构中有个 48 字节的字符数组 cb[]，可以根据不同网域或不同应用的需要来载送一些附加信息。在这里，用它来传递一个 unix_skb_parms 结构，其内容包括发送方的身份信息以及一个指向要授权接收方使用的已打开文件表的指针。

至此，报文中的数据都已经拷贝到了用户空间的缓冲区中，附加的信息也拷贝到了临时的 scm_cookie 结构中，从“接收”的角度来说，主要的任务已经完成了。但是，如果 sk_buff 结构中载送着对已打开文件的访问授权的话，那就还有点事要做。当一个进程把对它的若干已打开文件的访问权发送给另一个进程时，要把对这些文件访问权的描绘也接收下来。另一方面，如果这些已打开文件代表着 Unix 域插口，则发送者要为每个这样的已打开文件记下一笔账，说明对它的使用权正在传送的途中。而对方在接收到这些授权后则要负责“销账”。以前讲过，用户进程可以把调用参数 flags 中的 MSG_PEEK 标志置成 1，表示只是“偷看”一下接收队列中的第一个报文，而并不真的接收这个报文。所以 skb_recv_datagram()在这种情况下并不将报文从队列中摘除，而只是将该报文的共享计数 users 加 1 后便返回指向该 sk_buff 结构的指针。所以，在这两种情况下，对报文中载送的访问授权所作的处理也有所不同。如果接收方在调用 recv()等函数时的 MSG_PEEK 标志位为 0，也就是说正式“接收”了这些授权时，就要调用 unix_detach_fds()来销账。这个函数的代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1111 1121 ====================

[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>unix_detach_fds()]

```
static void unix_detach_fds(struct scm_cookie *scm, struct sk_buff *skb)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int i;

    scm->fp = UNIXCB(skb).fp;
    skb->destructor = sock_wfree;
    UNIXCB(skb).fp = NULL;

    for (i=scm->fp->count-1; i>=0; i--)
        unix_notinflight(scm->fp->fp[i]);
}
```

这里一方面将 sk_buff 结构中的 scm_fp_list 指针（也在前述的 cb 所载送的 unix_skb_parms 结构中）也拷贝到 scm_cookie 结构中，并为缓冲区的释放准备下一个函数 sock_wfree()。另一方面就通过 unix_notinflight()来销帐，表示该项授权已不再在“飞行中”。其代码在 net/unix/garbage.c 中：

==================== net/unix/garbage.c 130 137 ====================
[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>unix_detach_fds()>unix_notinflight()]

```
void unix_notinflight(struct file *fp)
{
    unix_socket *s=unix_get_socket(fp);
    if(s) {
        atomic_dec(&s->protinfo.af_unix.inflight);
        atomic_dec(&unix_tot_inflight);
    }
}
```

这里的 unix_get_socket()从给定的 file 结构出发找到其 inode 结构，如果该 inode 结构代表着一个 Unix 域插口，就返回指向其 sock 结构（inode 结构的一部分）的指针，代码亦在 net/unix/garbage.c 中：

==================== net/unix/garbage.c 95 114 ====================
[sys_socketcall()>sys_recvmsg()>unix_dgram_recvmsg()>unix_detach_fds()>unix_notinflight()>unix_get_socket()]

```
extern inline unix_socket *unix_get_socket(struct file *filp)
{
    unix_socket * u_sock = NULL;
    struct inode *inode = filp->f_dentry->d_inode;

    /*
     *  Socket ?
     */
    if (inode->i_sock) {
        struct socket * sock = &inode->u.socket_i;
        struct sock * s = sock->sk;

        /*
         *  PF_UNIX ?
         */
        if (s && sock->ops && sock->ops->family == PF_UNIX)
            u_sock = s;
    }
    return u_sock;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

如果接收方只是看一下，而并不真正接收呢？代码的作者在 unix_dgram_recvmsg()的代码中加了注解，说 POSIX 1003.1g 对此没有作出规定，所以这里采取了为 scm_cookie 结构复制一份 scm_fp_list 结构（而不是其指针）的做法。当然，这时候不能把账销掉，因为对这些已打开文件的访问权仍在递交的途中。

最后，将 sk_buff 结构所占的缓冲区释放掉（1457 行）。

回到 sock_recvmsg()的代码中（521 行），下一件事就是对接收到的附加信息（如果有的话）的处理了。inline 函数 scm_recv()的代码在 include/net/scm.h 中：

==================== include/net/scm.h 45 63 ====================

[sys_socketcall()>sys_recvfrom()>sock_recvmsg()>scm_recv()]

```
45  static __inline__ void scm_recv(struct socket *sock, struct msghdr *msg,
46                      struct scm_cookie *scm, int flags)
47  {
48      if (!msg->msg_control)
49      {
50          if (sock->passcred || scm->fp)
51              msg->msg_flags |= MSG_CTRUNC;
52          scm_destroy(scm);
53          return;
54      }
55
56      if (sock->passcred)
57          put_cmsg(msg, SOL_SOCKET, SCM_CREDENTIALS, sizeof(scm->creds), &scm->creds);
58
59      if (!scm->fp)
60          return;
61
62      scm_detach_fds(msg, scm);
63  }
```

如果接收进程在其 msghdr 结构中没有为控制信息（附加信息）准备好缓冲区，那就没有什么事可干了，所以把 scm_cookie 结构“销毁”就完事了（52 行）。所谓把 scm_cookie 结构“销毁”，实际上是指把依附于这个结构的缓冲区释放。至于 scm_cookie 结构本身，是作为 sock_recvmsg()的局部量存在于堆栈中，所以不存在释放空间的问题。这里要指出，只有在通过 recvmsg()进入 sys_recvmsg()时，才有可能在 msghdr 结构中为控制信息准备缓冲区，因为这个函数的参数之一是个 msghdr 结构指针；其它的函数都在程序中固定将 msghdr 结构内的指针 msg_control 设成 NULL。所以，只有通过 recvmsg()才能接收到对方的身份消息，也只有通过 recvmsg()才能接收对方传递过来的已打开文件的访问授权。

函数 put_cmsg()将有关的附加信息（在这里是有关对方身份的信息）递交到用户空间去，其代码在 net/coretscm.c 中，有兴趣的读者可自行阅读。

如果在 scm_cookie 结构中包含有对已打开文件的访问授权，那就要把这些已打开文件（由发送进程打开）纳入到接收进程的已打开文件表中。相对来说，这就要复杂一点了，scm_detach_fds()的代码也在 net/core/scm.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== net/core/scm.c 203 260 ====================

[sys_socketcall()>sys_recvfrom()>sock_recvmsg()>scm_recv()>scm_detach_fds()]

```c
void scm_detach_fds(struct msghdr *msg, struct scm_cookie *scm)
{
	struct cmsghdr *cm = (struct cmsghdr*)msg->msg_control;

	int fdmax = 0;
	int fdnum = scm->fp->count;
	struct file **fp = scm->fp->fp;
	int *cmfptr;
	int err = 0, i;

	if (msg->msg_controllen > sizeof(struct cmsghdr))
		fdmax = ((msg->msg_controllen - sizeof(struct cmsghdr))
			 / sizeof(int));

	if (fdnum < fdmax)
		fdmax = fdnum;

	for (i=0, cmfptr=(int*)CMSG_DATA(cm); i<fdmax; i++, cmfptr++)
	{
		int new_fd;
		err = get_unused_fd();
		if (err < 0)
			break;
		new_fd = err;
		err = put_user(new_fd, cmfptr);
		if (err) {
			put_unused_fd(new_fd);
			break;
		}
		/* Bump the usage count and install the file. */
		get_file(fp[i]);
		fd_install(new_fd, fp[i]);
	}

	if (i > 0)
	{
		int cmlen = CMSG_LEN(i*sizeof(int));
		if (!err)
			err = put_user(SOL_SOCKET, &cm->cmsg_level);
		if (!err)
			err = put_user(SCM_RIGHTS, &cm->cmsg_type);
		if (!err)
			err = put_user(cmlen, &cm->cmsg_len);
		if (!err) {
			cmlen = CMSG_SPACE(i*sizeof(int));
			msg->msg_control += cmlen;
			msg->msg_controllen -= cmlen;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
250             }
251         }
252         if (i < fdnum || (fdnum && fdmax <= 0))
253             msg->msg_flags |= MSG_CTRUNC;
254
255         /*
256          * All of the files that fit in the message have had their
257          * usage counts incremented, so we just free the list.
258          */
259         __scm_destroy(scm);
260 }
```

代码中的 fdmax 表示接收方 msghdr 结构中用来接收这些已打开文件指针的缓冲区容量；而 fdnum 则是通过报文传递过来的已打开文件指针的个数，此项信息已经在一个临时的 scm_cookie 数据结构中。显然，这两个数值只能以较小者为准。读者在前面已经看到过，在 scm_cookie 结构中有个指针 fp，指向一个 scm_fp_list 结构，而这个结构中则有一个 file 结构指针数组，数组中的每个元素都是指针，指向发送进程的一个已打开文件的 file 结构。现在，发送方已经将这个指针传过来了，接收方要做的就是把它“安装”到它自己的已打开文件表中。这样做了以后，两个进程就可以共享这个已打开文件了。当然，共享的双方必须在同一台计算机上。

代码中从 220 行开始的 for 循环就是对传过来并且在缓冲区容量之内的每个指针做这件事。首先当然是在当前进程的打开文件表中找到一个空闲位置，其下标即是新的打开文件号 new_fd。然后，宏操作 put_user()把这个新的打开文件号写入到用户空间中准备好的缓冲区中（msg->msg_control 指向这块缓冲区）。前面我们讲到缓冲区的容量问题，就是指用户空间中的这块缓冲区是否足够于用来返回这些新的打开文件号。一旦把传过来的指针安装在接收进程的打开文件表中，相应的 file 结构就多了一个用户，所以要通过 get_file()递增其共享计数。最后，就是把这个指针“安装”到接收进程（就是当前进程）的打开文件表中了，而 new_fd 则指明了在表中的位置。

前面讲过，msg->msg_control 指向用户空间的一块缓冲区，这块缓冲区实际上是由一个 cmsghdr 结构加上一个数据部分（用来返回新增的打开文件号）构成的。只要接收并安装了至少一个新的已打开文件，就要设置 cmsghdr 结构中的头部信息，包括实际使用的数据部分的长度 cmlen。最后，还要调用__scm_destroy()，将依附于 scm_cookie 结构的动态分配的存储空间（在这里是 scm_fp_list 结构）释放掉。

进程间对已打开文件的这种访问授权，在某些应用中是很有意义的。以插口的使用为例，我们以前讲过，“有连接”插口的典型运用是：在 accept()产生了一个新的已经建立起连接的插口以后，就 fork()一个子进程，让子进程去为 client 方提供服务，而父进程本身则又进入 accept()等待新的连接请求。不管怎么说，动态地 fork()一个子进程的代价终究还是不小的。现在有了跨进程的访问授权，就可以预先创建若干个子进程，一旦 accept()产生了一个新的插口，就可以把对该插口的访问权传给某个空闲的子进程，让它去提供服务。这样，就可以省去了因每次都 fork()一个新进程而引起的延迟。

在上面这个情景中，我们假定在接收方的 receive_queue 队列中已经有报文在等待，所以接收进程不需要在 skb_recv_datagram()中睡眠等待。如果接收队列中没有报文，那接收进程就需要睡眠等待，等到有报文到达时才被唤醒，这种情景读者在前面讲述 accept()和 connect()时已经看到过了。

再来看 Unix 域“无连接”插口的报文发送。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

与 sock_recvmsg()相对应的函数是 sock_sendmsg()，其代码在文件 net/socket.c 中：

==================== net/socket.c 501 512 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()]
```
int sock_sendmsg(struct socket *sock, struct msghdr *msg, int size)
{
	int err;
	struct scm_cookie scm;

	err = scm_send(sock, msg, &scm);
	if (err >= 0) {
		err = sock->ops->sendmsg(sock, msg, size, &scm);
		scm_destroy(&scm);
	}
	return err;
}
```

首先是对发送者身份以及附加控制信息的处理，inline 函数 scm_send()的代码在 include/net/scm.h 中。与接收报文时相似，只有在通过 sendmsg()进入 sys_sendmsg()时才有可能随同报文发送这些附加信息，因为这个函数的参数之一是个 msghdr 结构指针，其它的函数都在程序中固定将 msghdr 结构内的指针 msg_control 设成 NULL。

==================== include/net/scm.h 33 43 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()]
```
static __inline__ int scm_send(struct socket *sock, struct msghdr *msg,
			       struct scm_cookie *scm)
{
	memset(scm, 0, sizeof(*scm));
	scm->creds.uid = current->uid;
	scm->creds.gid = current->gid;
	scm->creds.pid = current->pid;
	if (msg->msg_controllen <= 0)
		return 0;
	return __scm_send(sock, msg, scm);
}
```

==================== net/core/scm.c 114 169 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>__scm_send()]
```
int __scm_send(struct socket *sock, struct msghdr *msg, struct scm_cookie *p)
{
	struct cmsghdr *cmsg;
	int err;

	for (cmsg = CMSG_FIRSTHDR(msg); cmsg; cmsg = CMSG_NXTHDR(msg, cmsg))
	{
		err = -EINVAL;

		/* Verify that cmsg_len is at least sizeof(struct cmsghdr) */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            /* The first check was omitted in <= 2.2.5. The reasoning was
               that parser checks cmsg_len in any case, so that
               additional check would be work duplication.
               But if cmsg_level is not SOL_SOCKET, we do not check
               for too short ancillary data object at all! Oops.
               OK, let's add it...
             */
            if (cmsg->cmsg_len < sizeof(struct cmsghdr) ||
                (unsigned long)(((char*)cmsg - (char*)msg->msg_control)
                         + cmsg->cmsg_len) > msg->msg_controllen)
                goto error;

            if (cmsg->cmsg_level != SOL_SOCKET)
                continue;

            switch (cmsg->cmsg_type)
            {
            case SCM_RIGHTS:
                err=scm_fp_copy(cmsg, &p->fp);
                if (err<0)
                    goto error;
                break;
            case SCM_CREDENTIALS:
                if (cmsg->cmsg_len != CMSG_LEN(sizeof(struct ucred)))
                    goto error;
                memcpy(&p->creds, CMSG_DATA(cmsg), sizeof(struct ucred));
                err = scm_check_creds(&p->creds);
                if (err)
                    goto error;
                break;
            default:
                goto error;
            }
        }

        if (p->fp && !p->fp->count)
        {
            kfree(p->fp);
            p->fp = NULL;
        }
        return 0;

error:
        scm_destroy(p);
        return err;
}
```

读者可以对照前面的 scm_recv()自行阅读这段代码。

把附加信息搭挂到报文上以后，就要通过具体的发送函数把报文发送出去了。对于 Unix 域的无连

PDF-XCHANGE Click to buy NOW! www.docu-track.com

接插口，这个函数是 unix_dgram_sendmsg()。这个函数比较大，我们分段阅读：

==================== net/unix/af_unix.c 1145 1185 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()]

```
/*
 *   Send AF_UNIX data.
 */

static int unix_dgram_sendmsg(struct socket *sock, struct msghdr *msg, int len,
                              struct scm_cookie *scm)
{
     struct sock *sk = sock->sk;
     struct sockaddr_un *sunaddr=msg->msg_name;
     unix_socket *other = NULL;
     int namelen = 0; /* fake GCC */
     int err;
     unsigned hash;
     struct sk_buff *skb;
     long timeo;

     err = -EOPNOTSUPP;
     if (msg->msg_flags&MSG_OOB)
          goto out;

     if (msg->msg_namelen) {
          err = unix_mkname(sunaddr, msg->msg_namelen, &hash);
          if (err < 0)
               goto out;
          namelen = err;
     } else {
          sunaddr = NULL;
          err = -ENOTCONN;
          other = unix_peer_get(sk);
          if (!other)
               goto out;
     }

     if (sock->passcred && !sk->protinfo.af_unix.addr &&
         (err = unix_autobind(sock)) != 0)
          goto out;

     err = -EMSGSIZE;
     if ((unsigned)len > sk->sndbuf - 32)
          goto out;

```

首先，“无连接”插口不支持“编外”的 OOB 报文，也不支持除 MSG_DONTWAIT 和 MSG_NOSIGNAL 以外的任何标志位。如果在调用参数中提供了对方地址，那就先通过 unix_mkname() 将其“规格化”后备用。如果没有提供对方地址呢？那就是假定已经调用过 connect()，设置好了通向

目标插口的路径，所以通过 unix_peer_get()就可以获得指向对方 sock 结构的指针。可是，要是实际上并没有调用过 connect()呢？那当然不能再往前走了，所以此时的出错代码为-ENOTCONN。此外，如果插口的可选项 passcred 规定要随报文传递与插口身份有关的信息（注意与发送进程身份的区别），而又从未通过 bind()为其指定一个地址，那就要通过 unix_autobind()为其自动生成一个。插口的发送报文缓冲区大小记录在其 sock 结构的 sndbuf 字段中，但是要保留 32 字节用于控制目的，所以 1183 行据此检查报文的长度。这里要注意，虽然在报文发送之前和接收之后可以采用 iovec[]把报文分散存放在多个缓冲区中，但是在发送的过程中总是在同一个缓冲区中。这一来是对网络报文的模拟，二来也是因为通过 iovec[]提供的缓冲区是在用户空间，而不是在系统空间。我们继续往下看：

==================== net/unix/af_unix.c 1186 1200 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()]

```
1186      skb = sock_alloc_send_skb(sk, len, 0, msg->msg_flags&MSG_DONTWAIT, &err);
1187      if (skb==NULL)
1188           goto out;
1189
1190      memcpy(UNIXCREDS(skb), &scm->creds, sizeof(struct ucred));
1191      if (scm->fp)
1192           unix_attach_fds(scm, skb);
1193
1194      skb->h.raw = skb->data;
1195      err = memcpy_fromiovec(skb_put(skb,len), msg->msg_iov, len);
1196      if (err)
1197           goto out_free;
1198
1199      timeo = sock_sndtimeo(sk, msg->msg_flags & MSG_DONTWAIT);
1200
```

接着，就是为报文的发送分配一个 sk_buff 结构，包括所需的缓冲区。函数 sock_sendmsg()在通过插口 proto_ops 的结构调用 unix_dgram_sendmsg()之前，先组装了一个临时的 scm_cookie 结构，其中的信息来自更上层的已组装、或者发送进程作为参数传递过来的 msghdr 结构中。现在载送报文的 sk_buff 结构和缓冲区已经分配好，所以把这些信息先拷贝进去（见 1190～1195 行）。这里的 unix_attach_fds()）处理对已打开文件的访问授权，前面已经讲到过了，其代码也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1135 1143 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()>unix_attach_fds()]

```
static void unix_attach_fds(struct scm_cookie *scm, struct sk_buff *skb)
{
     int i;
     for (i=scm->fp->count-1; i>=0; i--)
          unix_inflight(scm->fp->fp[i]);
     UNIXCB(skb).fp = scm->fp;
     skb->destructor = unix_destruct_fds;
     scm->fp = NULL;
}
```

同样，如果所授权的已打开文件代表着 Unix 域插口，则要在传递过程中记下一笔账。读者可结合

PDF-XCHANGE Click to buy NOW! www.docu-track.com

前面的 unix_notinflight()阅读 unix_inflight()的代码（net/unix/garbage.c）：

==================== net/unix/garbage.c 116 128 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()>unix_attach_fds()>unix_inflight()]

```
116  /*
117   *   Keep the number of times in flight count for the file
118   *   descriptor if it is for an AF_UNIX socket.
119   */
120
121  void unix_inflight(struct file *fp)
122  {
123       unix_socket *s=unix_get_socket(fp);
124       if(s) {
125            atomic_inc(&s->protinfo.af_unix.inflight);
126            atomic_inc(&unix_tot_inflight);
127       }
128  }
```

然后，再将报文中的数据从用户空间的缓冲区中拷贝到依附于 sk_buff 结构的缓冲区中。根据 msghdr 结构中的具体设置，用户空间中的数据有可能分散在若干个缓冲区中，但是依附于 sk_buff 结构的缓冲区却只有一个，其大小应是用户空间中各个缓冲区中数据长度的总和。函数 memcpy_fromiovec()与我们前面看到过的 memcpy_toiovec()相似，只是拷贝的方向相反。

此外，inline 函数 skb_put()，根据数据的总长度调整 sk_buff 结构中的指针 skb->tail 和当前数据长度 skb->len，而返回 skb->tail 的初值，也就是缓冲区的起点（1195 行），这段代码在 include/linux/skbuff.h 中：

==================== include/linux/skbuff.h 694 713 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()>skb_put()]

```
694  /**
695   *   skb_put - add data to a buffer
696   *   @skb: buffer to use
697   *   @len: amount of data to add
698   *
699   *   This function extends the used data area of the buffer. If this would
700   *   exceed the total buffer size the kernel will panic. A pointer to the
701   *   first byte of the extra data is returned.
702   */
703
704  static inline unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
705  {
706       unsigned char *tmp=skb->tail;
707       skb->tail+=len;
708       skb->len+=len;
709       if(skb->tail>skb->end) {
710            skb_over_panic(skb, len, current_text_addr());
711       }
712       return tmp;
713  }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

最后，还要通过一个 inline 函数 sock_sndtimeo()确定对发送过程的时间限制：

==================== include/net/sock.h 1249 1252 ====================

```
static inline long sock_sndtimeo(struct sock *sk, int noblock)
{
	return noblock ? 0 : sk->sndtimeo;
}
```

所有这些准备工作都完成以后，就要把载送着报文的 sk_buff 结构送到目标插口一侧去了。我们继续往下看：

==================== net/unix/af_unix.c 1201 1247 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()]

```
restart:
	if (!other) {
		err = -ECONNRESET;
		if (sunaddr == NULL)
			goto out_free;

		other = unix_find_other(sunaddr, namelen, sk->type, hash, &err);
		if (other==NULL)
			goto out_free;
	}

	unix_state_rlock(other);
	err = -EPERM;
	if (!unix_may_send(sk, other))
		goto out_unlock;

	if (other->dead) {
		/*
		 *	Check with 1003.1g - what should
		 *	datagram error
		 */
		unix_state_runlock(other);
		sock_put(other);

		err = 0;
		unix_state_wlock(sk);
		if (unix_peer(sk) == other) {
			unix_peer(sk)=NULL;
			unix_state_wunlock(sk);

			unix_dgram_disconnected(sk, other);
			sock_put(other);
			err = -ECONNREFUSED;
		} else {
			unix_state_wunlock(sk);
		}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1237
1238            other = NULL;
1239            if (err)
1240                goto out_free;
1241            goto restart;
1242        }
1243
1244        err = -EPIPE;
1245        if (other->shutdown&RCV_SHUTDOWN)
1246            goto out_unlock;
1247
```

如果插口在本次发送之前已经通过 connect()建立了通向目标插口的路径，那么此时指针 other 已经指向对方的 sock 结构，否则即为 NULL。或者，虽然以前已经调用过 connect()，但是在本次发送时又给定了一个地址，此时 other 也是 0（见 1165 行）。所以，指针 other 为 NULL 时需要临时通过 unix_find_otherr()找到对方的 sock 结构。这个函数我们在 connect()一节中已经看到过了。找到了对方以后，还要通过 unix_may_send()测试一下是否允许从我方插口向对方插口发送，测试的准则是对方并没有通过 connect()把路径设置成通向“第三者”，unix_may_send()的代码我们也已经在讲述“无连接 ”插口的 connect()时看到过了。

虽然找到了对方插口，但是这个插口却并不一定还活着。我们在讲述“有连接”插口的 connect()时已经讨论过这方面的问题，回过头去看一下会有助于理解 1217～1242 行的这一段代码。在寻找对方插口的 sock 结构的过程中，函数 unix_find_other()和 unix_peer_get()是二者必经其一的，而这二者都直接或间接地调用 sock_hold()递增了对方插口的使用计数。所以 1223 行的 sock_put()就是与这二者之一配对的，也就是将对方插口的使用计数递减，如果减后达到了 0 就将其 sock 结构所占的空间释放。如果本次发送的目标只是临时的，那么这就可以了，下面就转到 restart 处看看是否还能找到具有相同地址的其它插口（原来的插口撤销后可能又重新建立了），不过通常此时 unix_find_other()会失败而出错返回。可是，如果这个目标是以前通过 connect()设置的，那就又不一样了。首先是还要再调用一次 sock_put()，因为当初在建立起通向目标的路径时也曾对目标插口的 sock 结构调用过一次 sock_hold()，现在要把它抵消。其次，既然是原先通过 connect()建立的比较稳固的伙伴关系（虽然是无连接模式），而现在对方已经“去世”，那就不像对临时给定的地址一样需要再找找有无相同地址的插口了，所以直接就将出错代码设置成-ECONNREFUSED，并转至 out_free 处返回。

现在只剩下最后一个可能的障碍了，那就是目标插口可能已经通过 shutdown()将接收报文的功能关闭了（但是插口并未撤销）。注意在这种情况下返回的出错代码为-EPIPE。

至此，所有的关卡都已经通过了，下面就要将报文（即 sk_buff 结构）挂入目标插口的 sock 结构里的接收队列中。

再在 unix_dgram_sendmsg()中继续往下看（net/unix/af_unix.c）

==================== net/unix/af_unix.c 1248 1278 ====================
[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_dgram_sendmsg()]

```
1248        if (unix_peer(other) != sk &&
1249            skb_queue_len(&other->receive_queue) > other->max_ack_backlog) {
1250            if (!timeo) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1251                err = -EAGAIN;
1252                goto out_unlock;
1253            }
1254
1255            timeo = unix_wait_for_peer(other, timeo);
1256
1257            err = sock_intr_errno(timeo);
1258            if (signal_pending(current))
1259                goto out_free;
1260
1261            goto restart;
1262        }
1263
1264        skb_queue_tail(&other->receive_queue, skb);
1265        unix_state_runlock(other);
1266        other->data_ready(other, len);
1267        sock_put(other);
1268        return len;
1269
1270 out_unlock:
1271        unix_state_runlock(other);
1272 out_free:
1273        kfree_skb(skb);
1274 out:
1275        if (other)
1276            sock_put(other);
1277        return err;
1278 }
```

下面就是将一个“无连接”模式报文挂入接收队列的过程，这与“有连接”模式中将一个“连接请求”报文挂入 server 插口接收队列的过程很相似。所以，我们这里就不重复了，读者可以跟前面 sys_connect()的代码对照阅读。

看完了“无连接”模式的报文接收与发送，我们再来看看“有连接”模式下的报文接收与发送。

与 unix_dgram_recvmsg()相对应，Unix 域“有连接”模式的报文接收是通过 unix_stream_recvmsg()完成的。这个函数的代码也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1499 1522 ====================

[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()]

```
1499 static int unix_stream_recvmsg(struct socket *sock, struct msghdr *msg, int size,
1500                       int flags, struct scm_cookie *scm)
1501 {
1502        struct sock *sk = sock->sk;
1503        struct sockaddr_un *sunaddr=msg->msg_name;
1504        int copied = 0;
1505        int check_creds = 0;
1506        int target;
1507        int err = 0;
1508        long timeo;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1509
1510     err = -EINVAL;
1511     if (sk->state != TCP_ESTABLISHED)
1512          goto out;
1513
1514     err = -EOPNOTSUPP;
1515     if (flags&MSG_OOB)
1516          goto out;
1517
1518     target = sock_rcvlowat(sk, flags&MSG_WAITALL, size);
1519     timeo = sock_rcvtimeo(sk, flags&MSG_DONTWAIT);
1520
1521     msg->msg_namelen = 0;
1522
```

以前讲过，“无连接”模式的插口是“无状态”的，而“有连接”模式的插口则是“有状态”的，其 sock 结构中的 state 就是用来实现一种“有限状态机”。所以这里要检查对方插口是否处于可以接收数据报文（而不是连接请求）的状态，实际上就是要检查连接是否已经建立好。

以前我们也讲过，“有连接”模式的通信支持所谓“编外”，即 OOB 报文的传递，这种报文在传递中具有较高的优先级，但是，那只是就一般而言，它实际上是为网络环境，特别是速度比较慢的网络环境而设计的。对 Unix 域的插口来说，收发双方都有同一台机器上实际上并没有这种需要，所以 Unix 域的“有连接”插口并不支持这种报文。

调用参数中的 size 表示接收缓冲区的大小，也就是想要接收的字节数。如前所述，“有连接”模式的接收是跨越报文边界的。我们以前曾通过一个例子来说明，当接收队列中有两个报文，而第一个报文中的数据并没有将接收缓冲区填满时，就会继续从第二个报文中读取数据。可是，当时我们没有进一步说明，如果队列中只有一个报文，而又不能将接收缓冲区填满的话，那又当如何？这时候接收进程是继续等待新报文的到来，还是返回一个只是部分填满的缓冲区？现在就要回答这个问题了。这里在代码中有个局部量 target，就是表示对数据量的最低要求。如果接收队列中已经没有报文了，但是已经接收到缓冲区中的字节数还低于这个最低要求，那就要睡眠等待，否则就返回了（虽然不无遗憾）。通常，这个最低要求是 1。但是接收进程可以在参数 flags 中设置一个 MSG_WAITALL，表示“不达目的决不收兵”。所以在 1518 行中通过 sock_rcvlowat()决定这个数值，然后设置当前的目标 target。

==================== include/net/sock.h 1254 1257 ====================

[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()>sock_rcvlowat()]

```
1254 static inline int sock_rcvlowat(struct sock *sk, int waitall, int len)
1255 {
1256     return (waitall ? len : min(sk->rcvlowat, len)) ? : 1;
1257 }
```

另一方面，也要通过 sock_rcvtimeo()确定对发送过程的时间限制，这我们已经在前面看到过了。再继续往下看：

==================== net/unix/af_unix.c 1523 1628 ====================

[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()]

```
1523     /* Lock the socket to prevent queue disordering
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * while sleeps in memcpy_tomsg
 */

down(&sk->protinfo.af_unix.readsem);

do
{
    int chunk;
    struct sk_buff *skb;

    skb=skb_dequeue(&sk->receive_queue);
    if (skb==NULL)
    {
        if (copied >= target)
            break;

        /*
         *   POSIX 1003.1g mandates this order.
         */

        if ((err = sock_error(sk)) != 0)
            break;
        if (sk->shutdown & RCV_SHUTDOWN)
            break;
        err = -EAGAIN;
        if (!timeo)
            break;
        up(&sk->protinfo.af_unix.readsem);

        timeo = unix_stream_data_wait(sk, timeo);

        if (signal_pending(current)) {
            err = sock_intr_errno(timeo);
            goto out;
        }
        down(&sk->protinfo.af_unix.readsem);
        continue;
    }

    if (check_creds) {
        /* Never glue messages from different writers */
        if (memcmp(UNIXCREDS(skb), &scm->creds, sizeof(scm->creds)) != 0) {
            skb_queue_head(&sk->receive_queue, skb);
            break;
        }
    } else {
        /* Copy credentials */
        scm->creds = *UNIXCREDS(skb);
        check_creds = 1;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        }

        /* Copy address just once */
        if (sunaddr)
        {
            unix_copy_addr(msg, skb->sk);
            sunaddr = NULL;
        }

        chunk = min(skb->len, size);
        if (memcpy_toiovec(msg->msg_iov, skb->data, chunk)) {
            skb_queue_head(&sk->receive_queue, skb);
            if (copied == 0)
                copied = -EFAULT;
            break;
        }
        copied += chunk;
        size -= chunk;

        /* Mark read part of skb as used */
        if (!(flags & MSG_PEEK))
        {
            skb_pull(skb, chunk);

            if (UNIXCB(skb).fp)
                unix_detach_fds(scm, skb);

            /* put the skb back if we didn't use it up.. */
            if (skb->len)
            {
                skb_queue_head(&sk->receive_queue, skb);
                break;
            }

            kfree_skb(skb);

            if (scm->fp)
                break;
        }
        else
        {
            /* It is questionable, see note in unix_dgram_recvmsg.
             */
            if (UNIXCB(skb).fp)
                scm->fp = scm_fp_dup(UNIXCB(skb).fp);

            /* put message back and return */
            skb_queue_head(&sk->receive_queue, skb);
            break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1622            }
1623        } while (size);
1624
1625        up(&sk->protinfo.af_unix.readsem);
1626 out:
1627        return copied ? : err;
1628 }
```

读者已经看到，这里的主体是一个 do-while 循环。在循环中，每次从接收队列中摘下一个 sk_buff 结构，然后就从报文中读取数据和附加信息，直至缓冲区已经填满（size 变成了 0）；或者接收队列中已经没有报文，而接收的最低要求又已满足（见 1538 行）。如果缓冲区已经填满而报文中尚有数据剩余则将剩余部分退回接收队列（1603 行），以备下一次再来继续接收。反之，如果缓冲区未满而接收队列中已经没有报文，则视最低要求是否已经满足而决定是睡眠等待还是返回。这整个过程涉及队列操作，所以不允许受到打扰。同时，所需的时间又相对较长，不能单纯地靠加锁（spin_lock）来保护，所以需要通过内核信号量加以保护，这个内核信号量就在目标插口的 sock 结构中(见 1527、1551、1559 以及 1625 行)。

循环体中所调用的函数以及其它各种“要素”大部分都已经在前面出现过，所以我们基本上把这一段代码留给读者作为练习，只是对其中的几个函数及变量略作说明。

首先，inline 函数 skb_dequeue()从给定队列（在这里是插口的接收队列）的前端摘下一个报文，即 sk_buff 数据结构（见 1534 行）。如果调用参数 flags 中的 MSG_PEEK 为 1，则在从报文中读取了数据和附加信息以后要通过 skb_queue_head()将已经脱链的报文退还到接收队列中的前端（见 1620 行）。要注意的是，这种“偷看”以一个报文为限。也就是说，不管报文缓冲区多大，也不管最低要求的数值 target 多大，看了从队列中摘下的第一个报文并将它退回以后就结束了（见 1621 行）。这样的处理使程序得以简化（否则还得准备下一个临时的队列），道理上也讲得过去（因为是“偷看”嘛）。

另外，还有几个地方也要用到 skb_queue_head()。一是当接收缓冲区已经填满而 sk_buff 中的数据尚有剩余时（1603 行），这里的 skb->len 为报文中剩余的数据量；二是当通过 memcpy_toiovec()将报文中的数据拷贝到用户空间中的缓冲区时出了错（见 1583～1588 行）。例如，要是接收进程的页面映射表中缓冲区所在的页面为“写保护”，那当然就失败了，此时 memcpy_toiovec()返回-EFAULT，而正常则返回 0。不过这并不一定意味着整个接收操作的失败，因为有可能在此之前已经成功地接收了一些数据。所以，此时要看变量 copied 的数值，它反映着已经将多少数据拷贝到了用户空间。还有一种情况，就是接收缓冲区尚未填满，所以就企图从队列中的下一个报文中再取一些数据，可是却发现这下一个报文中的身份信息与前一个中的不同，也就是说下一个报文是来自另一个发送进程(见 1563～1568 行)。在这种情况下，当然不应该将来自两个不同进程的报文“粘贴”在一起，所以也要把下一个报文退还到接收队列中。

读者也许感到奇怪，既然是“有连接”模式，那就只有通过已经建立了连接的插口才能发送报文，怎么又可能会有来自其它进程的报文呢？其实很简单，连接是建立在两个插口之间，而不是两个进程之间的。拥有已经建立好连接的插口的进程，既可以 fork()出若干个子进程，也可以将对这个插口的访问通过 sendmsg()授权给另一个进程。

接收了一个报文并读取了所载送的信息以后，要通过 kfree_skb()释放报文的缓冲区（1607 行），这个 inline 函数的代码在 include/linux/skbuff.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/skbuff.h 209 226 ====================

[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()>kfree_skb()]

```
/*
 * If users==1, we are the only owner and are can avoid redundant
 * atomic change.
 */

/**
 *   kfree_skb - free an sk_buff
 *   @skb: buffer to free
 *
 *   Drop a reference to the buffer and free it if the usage count has
 *   hit zero.
 */

static inline void kfree_skb(struct sk_buff *skb)
{
     if (atomic_read(&skb->users) == 1 || atomic_dec_and_test(&skb->users))
          __kfree_skb(skb);
}
```

这里__kffee_skb()的代码在 net/core/skbuff.c 中：

==================== net/core/skbuff.c 272 293 ====================

[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()>kfree_skb()>__kfree_skb()]

```
void __kfree_skb(struct sk_buff *skb)
{
     if (skb->list) {
          printk(KERN_WARNING "Warning: kfree_skb passed an skb still "
                "on a list (from %p).\n", NET_CALLER(skb));
          BUG();
     }

     dst_release(skb->dst);
     if(skb->destructor) {
          if (in_irq()) {
               printk(KERN_WARNING "Warning: kfree_skb on hard IRQ %p\n",
                    NET_CALLER(skb));
          }
          skb->destructor(skb);
     }
#ifdef CONFIG_NETFILTER
     nf_conntrack_put(skb->nfct);
#endif
     skb_headerinit(skb, NULL, 0);  /* clean state */
     kfree_skbmem(skb);
}
```

这个函数也许比读者想像的要复杂。我们把这里调用的一些函数留给读者，只是要提请读者注意：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

如果 sk_buff 结构中的函数指针 destructor 为非 0，就要先调用这个函数。那么，具体到 unix_stream_recvmsg()，这个指针是什么呢？后面读者就会看到，这是 sock_wfree()，是由发送报文的一方安排好了的。

最后，就是“有连接”模式的报文发送了，这是由 net/unix/af_unix.c 中的函数 unix_stream_sendmsg()完成的。

==================== net/unix/af_unix.c 1281 1309 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_stream_sendmsg()]

```
1281 static int unix_stream_sendmsg(struct socket *sock, struct msghdr *msg, int len,
1282                      struct scm_cookie *scm)
1283 {
1284     struct sock *sk = sock->sk;
1285     unix_socket *other = NULL;
1286     struct sockaddr_un *sunaddr=msg->msg_name;
1287     int err,size;
1288     struct sk_buff *skb;
1289     int limit=0;
1290     int sent=0;
1291 
1292     err = -EOPNOTSUPP;
1293     if (msg->msg_flags&MSG_OOB)
1294         goto out_err;
1295 
1296     if (msg->msg_namelen) {
1297         err = (sk->state==TCP_ESTABLISHED ? -EISCONN : -EOPNOTSUPP);
1298         goto out_err;
1299     } else {
1300         sunaddr = NULL;
1301         err = -ENOTCONN;
1302         other = unix_peer_get(sk);
1303         if (!other)
1304             goto out_err;
1305     }
1306 
1307     if (sk->shutdown&SEND_SHUTDOWN)
1308         goto pipe_err;
1309 
```

如前所述，在 Unix 域中虽然是“有连接”模式也不支持 OOB 报文。并且除 MSG_DONTWAIT 和 MSG_NOSIGNAL 以外所有的标志位都不允许使用。另一方面，既然是“有连接”模式的发送，就不允许指定接收方地址。通过 unix_peer_get()取得对方插口的 sock 结构指针 other 以后，这里并没有检验它是否还活着，以及是否已经关闭了报文接收。这是为什么呢？下面读者就会看到，整个发送过程是通过一个循环来完成的，对这两个条件的检查在每次循环中都要进行，而不是只检查一次。不过，倒是要先检查一下发送方插口本身是否已经关闭报文发送（见 1307 行）。再往下看：

==================== net/unix/af_unix.c 1310 1385 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_stream_sendmsg()]

```
	while(sent < len)
	{
		/*
		 *	Optimisation for the fact that under 0.01% of X messages typically
		 *	need breaking up.
		 */

		size=len-sent;

		/* Keep two messages in the pipe so it schedules better */
		if (size > sk->sndbuf/2 - 16)
			size = sk->sndbuf/2 - 16;

		/*
		 *	Keep to page sized kmalloc()'s as various people
		 *	have suggested. Big mallocs stress the vm too
		 *	much.
		 */

		if (size > PAGE_SIZE-16)
			limit = PAGE_SIZE-16; /* Fall back to a page if we can't grab a big buffer this instant */
		else
			limit = 0;	/* Otherwise just grab and wait */

		/*
		 *	Grab a buffer
		 */

		skb=sock_alloc_send_skb(sk,size,limit,msg->msg_flags&MSG_DONTWAIT, &err);

		if (skb==NULL)
			goto out_err;

		/*
		 *	If you pass two values to the sock_alloc_send_skb
		 *	it tries to grab the large buffer with GFP_BUFFER
		 *	(which can fail easily), and if it fails grab the
		 *	fallback size buffer which is under a page and will
		 *	succeed. [Alan]
		 */
		size = min(size, skb_tailroom(skb));

		memcpy(UNIXCREDS(skb), &scm->creds, sizeof(struct ucred));
		if (scm->fp)
			unix_attach_fds(scm, skb);

		if ((err = memcpy_fromiovec(skb_put(skb,size), msg->msg_iov, size)) != 0) {
			kfree_skb(skb);
```

```
            goto out_err;
        }

        unix_state_rlock(other);

        if (other->dead || (other->shutdown & RCV_SHUTDOWN))
            goto pipe_err_free;

        skb_queue_tail(&other->receive_queue, skb);
        unix_state_runlock(other);
        other->data_ready(other, size);
        sent+=size;
    }
    sock_put(other);
    return sent;

pipe_err_free:
    unix_state_runlock(other);
    kfree_skb(skb);
pipe_err:
    if (sent==0 && !(msg->msg_flags&MSG_NOSIGNAL))
        send_sig(SIGPIPE,current,0);
    err = -EPIPE;
out_err:
      if (other)
        sock_put(other);
    return sent ? : err;
}
```

为什么“有连接”模式的报文发送要通过一个循环来完成，而在“无连接”模式中就不需要循环呢？这又跟二者的语义有关。

“无连接”模式是面向报文的，必须维持报文的边界。用户（进程）交下来一块数据，不管多大（只要不超过极限）也要在一个报文中发送出去。至于用户交下来的数据块大小是否合理，那是用户的事。如果用户交下来的数据块太大，则立即失败返回。

而“有连接”模式就不同了。“有连接”模式是面向字节流的，所以可以在发送方将一个大报文分割成若干较小的报文来发送。这样比较有利于改善报文传递的效率，也减轻了用户的负担。那么，如果用户交下的报文太大，把它分割成多大才是合适的呢？或者说，多大的报文才需要分割呢？代码中把这个界线定为 sk->sndbuf 的一半（16 字节保留用于头部结构，在下面的讨论中我们都把它略去不计）。插口的 sock 结构中有个参数 sndbuf，是可以通过 setsockopt()设置和改变的。这个参数大致上决定了插口在正常条件下可以占用作为发送缓冲区的总的存储区大小。那么，为什么是这个值的一半呢？答案是，把它分成两半来用有利于提高效率。这样，就可以使得当发送方在准备后一半时，接收方已经在接收前一半了。通过对两个缓冲区的循环使用和流通，就可以形成一个“流水线”，从而提高效率。在网络环境下或者发送方和接收方在两个不同处理器上运行时，这种效果就尤为突出。可是，有时候由于系统中资源使用和周转的限制，这个值的一半也还是太大而一时分配不到所需的缓冲区，这时候怎么办呢？当然可以睡眠等待，但是更好的办法是退而求其次，分配一块再小一些的缓冲区，这样显然

更有利于资源的周转和在这种“逆境”下效率的提高。不过，小到一个页面以下就不合适了，因为以页面为单位分配空间比较简单，效率也较高。这就是代码第 1330 行将变量 limit 设置成 PAGE_SIZE-16 的道理。这个变量的作用，要与 sock_alloc_send_skb()的代码结合起来看才清楚。这个函数的代码在 net/core/sock.c 中：

==================== net/core/sock.c 741 814 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_stream_sendmsg()>sock_alloc_send_skb()]

```
/*
 *   Generic send/receive buffer handlers
 */

struct sk_buff *sock_alloc_send_skb(struct sock *sk, unsigned long size,
              unsigned long fallback, int noblock, int *errcode)
{
    int err;
    struct sk_buff *skb;
    long timeo;

    timeo = sock_sndtimeo(sk, noblock);

    while (1) {
        unsigned long try_size = size;

        err = sock_error(sk);
        if (err != 0)
            goto failure;

        /*
         *   We should send SIGPIPE in these cases according to
         *   1003.1g draft 6.4. If we (the user) did a shutdown()
         *   call however we should not.
         *
         *   Note: This routine isnt just used for datagrams and
         *   anyway some datagram protocols have a notion of
         *   close down.
         */

        err = -EPIPE;
        if (sk->shutdown&SEND_SHUTDOWN)
            goto failure;

        if (atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
            if (fallback) {
                /* The buffer get won't block, or use the atomic queue.
                * It does produce annoying no free page messages still.
                */
                skb = alloc_skb(size, GFP_BUFFER);
                if (skb)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
782                         break;
783                     try_size = fallback;
784                 }
785                 skb = alloc_skb(try_size, sk->allocation);
786                 if (skb)
787                     break;
788                 err = -ENOBUFS;
789                 goto failure;
790             }
791
792             /*
793              *   This means we have too many buffers for this socket already.
794              */
795
796             set_bit(SOCK_ASYNC_NOSPACE, &sk->socket->flags);
797             set_bit(SOCK_NOSPACE, &sk->socket->flags);
798             err = -EAGAIN;
799             if (!timeo)
800                 goto failure;
801             if (signal_pending(current))
802                 goto interrupted;
803             timeo = sock_wait_for_wmem(sk, timeo);
804         }
805
806         skb_set_owner_w(skb, sk);
807         return skb;
808
809 interrupted:
810         err = sock_intr_errno(timeo);
811 failure:
812         *errcode = err;
813         return NULL;
814 }
```

这个函数先试着按 size 大小分配空间（注意 755 行将 try_size 设置成 size），分配时的优先级别为 GFP_J3UFFER。如果行，就万事大吉了。不行的话，就要看参数 fallback。要是 fallback 为 0，那就或者通过 sock_wait_for_wmem()睡眠等待，或者失败返回，具体取决于参数 noblock。可是，要是 fallback 非 0 的话，那就再试着按 fallback 的大小再分配一次，这一次分配的优先级别则改成 sk->allocation，通常是比 GFP_BUFFER 高一些（事实上，在 sock_init_data()中将这个优先级设置成 GFP_KERNEL），而 fallback 则通常比 size 要小。这样，第二次分配成功的希望就相当大了。当然，还有可能再失败，那就要睡眠等待或失败返回了。换言之，当 fallback 为 0 时的空间分配是硬性的，成就成，不成就不成；而 fallback 为非 0 时的空间分配则是软性的，能按 size 的大小分配最好，不成就退而求其次，按 fallback 的大小进行分配。具体的缓冲区分配由 sock_wmalloc()进行，其代码也在 net/core/sock.c 中：

==================== net/core/sock.c 654 667 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_stream_sendmsg()>sock_alloc_send_skb()>sock_wmalloc()]

```
654 /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Allocate a skb from the socket's send buffer.
 */
struct sk_buff *sock_wmalloc(struct sock *sk, unsigned long size, int force, int priority)
{
     if (force || atomic_read(&sk->wmem_alloc) < sk->sndbuf) {
          struct sk_buff * skb = alloc_skb(size, priority);
          if (skb) {
               skb_set_owner_w(skb, sk);
               return skb;
          }
     }
     return NULL;
}
```

插口的 sock 结构中有个整数 wmem_alloc，记录着插口正占用着的缓冲区字节数。当分配缓冲区成功时就由这里的 skb_set_owner_w()将分配得的实际大小加到 wmem_alloc 中，而释放时则从中减去。这个函数虽然小，却起着很重要的作用，其代码在 include/net/sock.h 中：

==================== include/net/sock.h 1125 1140 ====================

[sys_socketcall()>sys_sendmsg()>sock_sendmsg()>unix_stream_sendmsg()>sock_alloc_send_skb()>sock_wmalloc()>skb_set_owner_w()]

```
/*
 *   Queue a received datagram if it will fit. Stream and sequenced
 *   protocols can't normally use this as they need to fit buffers in
 *   and play with them.
 *
 *   Inlined as it's very short and called for pretty much every
 *   packet ever received.
 */

static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
{
     sock_hold(sk);
     skb->sk = sk;
     skb->destructor = sock_wfree;
     atomic_add(skb->truesize, &sk->wmem_alloc);
}
```

注意这里将 sk_buff 结构中的函数指针 destructor 设置成指向 sock_wfree()。这样，报文的接收方在释放这个数据结构时就会先调用这个函数。

回到 unix_stream_sendmsg()的代码中的第 1338 行，这里的调用参数 limit 就是上面的 fallback。所以，当 size 小于一个页面时 limit 为 0，此时对缓冲区分配的要求是硬性的，因为不能比一个页面再小了。可是，当 size 大于一个页面时，则 limit 的大小为一个页面，此时就可以“讨价还价”了。读者不妨回过去看一下 unix_dgram_sendmsg()的代码，那里对缓冲区分配的要求是硬性的，这是因为“无连接 ”模式的报文发送不允许改变报文的边界。通过 sock_alloc_send_skb()分配了缓冲区以后，缓冲区的实际大小可以调用 skb_tailroom()来获取。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

从这里往下的代码，读者应该已经很熟悉了。不过，不知读者看了以后是否感觉到有些异样？我们在 unix_stream_connect()中和 unix_dgram_sendmsg()中都曾看到，在将报文挂入接收队列之前要测试队列长度；如果其长度达到了某个限额就要睡眠等待。可是，在这里却没有看到。这是为什么？难道“有连接”模式的报文发送就不存在这个问题吗？是的，Unix 域“有连接”模式的报文发送确实不存在使对方的接收队列过长的问题。我们知道，连接是建立于插口之间，而不是进程之间。同时，我们已经看到，一个插口可以用作发送缓冲区的空间大小是有限制的。当发送方插口将缓区用光（取决于其 sock 结构中的 sndbuf），也即全部挂入到接收方插口的接收队列中去以后，就再也分配不到缓冲区，而只好在 sock_alloc_send_skb()中睡眠等待了。等待什么呢？等待接收方进程从队列中接收报文而将缓冲区释放出来。那时候，接收方就会在释放报文的 sk_buff 结构是通过其函数指针调用 sock_wfree()，其代码在 net/core/sock.c 中：

==================== net/core/sock.c 631 642 ====================
[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()>kfree_skb()>__kfree_skb()>sock_wfree()]

```
/*
 * Write buffer destructor automatically called from kfree_skb.
 */
void sock_wfree(struct sk_buff *skb)
{
     struct sock *sk = skb->sk;

     /* In case it might be waiting for more memory. */
     atomic_sub(skb->truesize, &sk->wmem_alloc);
     sk->write_space(sk);
     sock_put(sk);
}
```

这里调整了 sk->wmem_alloc 的数值，因为发送方插口实际占用的缓冲区减少了。而 sock 结构中的另一个函数指针 write_space 则在创建插口时（见 unix_create1()的代码，482 行）设置成指向 unix_write_space()，这个函数将可能因分配不到报文缓冲区而将正在睡眠等待的发送方进程唤醒，其代码在 net/unix/af_unix.c 中：

==================== net/core/sock.c 631 642 ====================
[sys_socketcall()>sys_recvmsg()>unix_stream_recvmsg()>kfree_skb()>__kfree_skb()>sock_wfree()>unix_write_space()]

```
static void unix_write_space(struct sock *sk)
{
     read_lock(&sk->callback_lock);
     if (unix_writable(sk)) {
          if (sk->sleep && waitqueue_active(sk->sleep))
               wake_up_interruptible(sk->sleep);
          sk_wake_async(sk, 2, POLL_OUT);
     }
     read_unlock(&sk->callback_lock);
}
```

当睡眠中的发送方进程被唤醒，再次试图分配报文缓冲区时，就能成功了。

最后还要强调指出，发送方 / 接收方的关系与 client/server 两方的关系完全是两码事。前者是对一

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

个具体的报文而言的，而后者仅与连接的建立有关，一旦建立了连接以后，任何一方都可以是一个报文的发送方。

7.8 插口的关闭

插口的关闭，即撤销，是通过文件系统界面的 close 系统调用完成的。读者不妨回想一下前面讲过的数据结构之间的联系，看看从插口的打开文件号和当前进程的 task_struct 结构开始，怎样才能找到插口的文件操作跳转表（数据结构 socket_file_ops）。该结构中的函数指针 release 指向 sock_close()，其代码在 net/socket.c 中：

==================== net/socket.c 692 707 ====================

[sys_close()>flip_close()>fput()>sock_close()]

```
int sock_close(struct inode *inode, struct file *filp)
{
	/*
	 *  It was possible the inode is NULL we were
	 *  closing an unfinished socket.
	 */

	if (!inode)
	{
		printk(KERN_DEBUG "sock_close: NULL inode\n");
		return 0;
	}
	sock_fasync(-1, filp, 0);
	sock_release(socki_lookup(inode));
	return 0;
}
```

首先是对插口的 fasync_list 的处理。我们在讲述 connect()时讲到过，server 方进程可以异步地等待来自 client 插口的连接请求。实际上，连接请求只是一个特例，异步接收同样也适用于普通报文的接收。在 socket 结构中有个 fasync_list 队列，如果想要在一个 server 插口上异步等待连接请求，或者从一个插口异步地接收报文，就可以通过 ioctl()系统调用分配一个 fasync_struct 数据结构，并将其挂入插口的 fasync_list 队列。现在插口既然要撤销了，当然要把 fasync_list 队列中的 fasync_struct 结构全都释放掉。函数 sock_fasync()的代码也在 net/socket.c 中，由于代码比较简单，我们就不详述了。接着，看 sock_release()，其代码也在 net/socket.c 中：

==================== net/socket.c 476 499 ====================

[sys_close()>filp_close()>f_put()>sock_close()>sock_release()]

```
/**
 *	sock_release	-	close a socket
 *	@sock: socket to close
 *
 *	The socket is released from the protocol stack if it has a release
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   callback, and the inode is then released if the socket is bound to
 *   an inode not a file.
 */

void sock_release(struct socket *sock)
{
    if (sock->ops)
        sock->ops->release(sock);

    if (sock->fasync_list)
        printk(KERN_ERR "sock_release: fasync list not empty!\n");

    sockets_in_use[smp_processor_id()].counter--;
    if (!sock->file) {
        iput(sock->inode);
        return;
    }
    sock->file=NULL;
}
```

用于 unix 域插口的两个 proto_ops 结构，即 unix_stream_ops 和 unix_dgram_ops，其中的 release 指针都指向 unix_release()，其代码在 net/unix/af_unix.c 中给出：

==================== net/unix/af_unix.c 525 535 ====================
[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()]

```
static int unix_release(struct socket *sock)
{
    unix_socket *sk = sock->sk;

    if (!sk)
        return 0;

    sock->sk = NULL;

    return unix_release_sock (sk, 0);
}
```

函数 unix_release_sock()的代码也在同一文件 net/unix/af_unix.c 中，我们分两段来看：

==================== net/unix/af_unix.c 353 394 ====================
[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()]

```
static int unix_release_sock (unix_socket *sk, int embrion)
{
    struct dentry *dentry;
    struct vfsmount *mnt;
    unix_socket *skpair;
    struct sk_buff *skb;
    int state;

```

```
    unix_remove_socket(sk);

    /* Clear state */
    unix_state_wlock(sk);
    sock_orphan(sk);
    sk->shutdown = SHUTDOWN_MASK;
    dentry = sk->protinfo.af_unix.dentry;
    sk->protinfo.af_unix.dentry=NULL;
    mnt = sk->protinfo.af_unix.mnt;
    sk->protinfo.af_unix.mnt=NULL;
    state = sk->state;
    sk->state = TCP_CLOSE;
    unix_state_wunlock(sk);

    wake_up_interruptible_all(&sk->protinfo.af_unix.peer_wait);

    skpair=unix_peer(sk);

    if (skpair!=NULL) {
         if (sk->type==SOCK_STREAM) {
              unix_state_wlock(skpair);
              skpair->shutdown=SHUTDOWN_MASK;  /* No more writes*/
              if (!skb_queue_empty(&sk->receive_queue) || embrion)
                   skpair->err = ECONNRESET;
              unix_state_wunlock(skpair);
              skpair->state_change(skpair);
              read_lock(&skpair->callback_lock);
              sk_wake_async(skpair,1,POLL_HUP);
              read_unlock(&skpair->callback_lock);
         }
         sock_put(skpair); /* It may now die */
         unix_peer(sk) = NULL;
    }

```

以前我们讲过，内核中有个杂凑表 unix_socket_table[]，每个插口的 sock 结构都挂在杂凑表中的某一个队列中，而 Unix 域 sock 结构中的 protinfo.af_unix.list 则指向该队列的头部。现在，第一件事就是通过 unix_remove_socket()将插口的 sock 结构从杂凑表的队列中脱链，其代码在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 234 239 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_remove_socket()]

```

static __inline__ void unix_remove_socket(unix_socket *sk)
{
    write_lock(&unix_table_lock);
    __unix_remove_socket(sk);
    write_unlock(&unix_table_lock);
```

==================== net/unix/af_unix.c 203 218 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_remove_socket()>__unix_remove_socket()]

```
static void __unix_remove_socket(unix_socket *sk)
{
	unix_socket **list = sk->protinfo.af_unix.list;
	if (list) {
		if (sk->next)
			sk->next->prev = sk->prev;
		if (sk->prev)
			sk->prev->next = sk->next;
		if (*list == sk)
			*list = sk->next;
		sk->protinfo.af_unix.list = NULL;
		sk->prev = NULL;
		sk->next = NULL;
		__sock_put(sk);
	}
}
```

下面，就要通过 sock_orphan()改变 sock 结构中的一些状态信息以及些指针了（include/linux/net/sock.h）。

==================== include/net/sock.h 998 1012 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_remove_socket()>__unix_remove_socket()>sock_orphan()]

```
/* Detach socket from process context.
 * Announce socket dead, detach it from wait queue and inode.
 * Note that parent inode held reference count on this struct sock,
 * we do not release it in this function, because protocol
 * probably wants some additional cleanups or even continuing
 * to work with this socket (TCP).
 */
static inline void sock_orphan(struct sock *sk)
{
	write_lock_bh(&sk->callback_lock);
	sk->dead = 1;
	sk->socket = NULL;
	sk->sleep = NULL;
	write_unlock_bh(&sk->callback_lock);
}
```

首先就是把 dead 置成 1，表示这个插口已经死亡了，我们在前面的一些函数中曾多次看到对这个状态信息的检测。接着，就把 sock 结构中指向相应 socket 结构的指针设成了 NULL。这里 write_lock_bh()是个宏操作，定义于 include/linux/spinlock.h 中：

==================== include/linux/spinlock.h 20 20 ====================

```
#define write_lock_bh(lock)	do { local_bh_disable();	write_lock(lock); } while (0)
```

从代码中可以看到，这里加了两把锁。第一把锁是在上一层的 unix_release_sock()中通过

PDF-XCHANGE Click to buy NOW! www.docu-track.com

unix_state_wlock()加上的，此函数的定义在文件 net/unix/af_unix.c 中给出：

==================== include/net/af_unix.h 39 39 ====================

```
#define unix_state_wlock(s) write_lock(&(s)->protinfo.af_unix.lock)
```

每一把“锁”，在锁上时都可以有两个状态，即“读”和“写”。这里的锁是 sock 结构中的 protinfo.af_unix.lock。注意，write_lock()并不是“锁上防写”，而是“为写而锁”。凡是要改变 sock 结构中任何内容时，都要将这把锁锁上至“写”状态。同时，哪怕只是要读 sock 结构中的内容，也得要加锁，不是锁至“写”状态就是锁至“读”状态。这样，只要有一个进程在写，其它的进程（在不同的处理器上运行，下同）就都不能读（当然也不能写），向只能在 write_lock()或 read_lock()中等待解锁（并不睡眠！）。反之，只要至少有一个进程在读，其它的进程就不能写，而只好在 write_lock()中等待解锁，但是却可以顺利通过 read_lock()。只有这样，才能保证任何进程都不会在改变内容的中途从中读出。可是，改变 sock 结构内容的过程往往需要比较长的时间，而仅仅要读 sk->dead 的内容就没有必要等所有的改变都完成，所以就为这个目的专门设置了另外一把锁，那就是 sock 结构中的 callback_lock。凡是需要读 sk->dead 的内容时，就至少要锁上其中的一把，但是哪一把都可以，即“写”状态或“读”状态都可以。

完成了对 sock 结构的改变以后，就可以唤醒正在睡眠中等待着要使用该插口的进程了。这样的进程有两种，一种是等待着报文的到达，另一种则是等待着将报文挂入本插口的接收队列中。这些进程被唤醒以后都会再一次检测 sk->dead，当发现这个插口已经死亡时就会出错返回。

在讲述 connect()的时候，读者已经看到在 sock 结构中有个指针 pair，而在 net/unix/af_unix.c 中对函数 unix_peer()的定义就是：

==================== net/unix/af_unix.c 140 140 ====================

```
#define unix_peer(sk) ((sk)->pair)
```

已经建立起连接的两个“有连接”模式插口互相通过这个指针指向对方的 sock 结构。至于“无连接”模式的插口，则为之调用了 connect()的插口通过这个指针单向地指向对方的 sock 结构。同时，只要一个插口的这个指针非 0，则它就是对方插口的一个用户，而对方插口的用户计数就包含了这个插口，所以要通过 sock_put()递减对方的用户计数（见 391 行），并且将这个指针清 0。由于“有连接”模式插口的“连接”是双向的，此时还要对连接的对方也作一些处理（380～390 行）。首先要切断对方继续向我方发送报文的功能。其次，如果我方的接收队列中还有报文（在 unix_release()中调用 unix_release_sock()时的调用参数 embrion 为 0），就要告知对方，把对方 sock 结构中的出错代码 err 设成 ECONNRESET。这样，如果有进程正在系统调用中对其进行某些操作，就会出错返回，否则下一次为对方插口而进入系统调用时也会出错返回。内核代码中有个 inline 函数 sock_error()，定义于 include/net/sock.h 中：

==================== include/net/sock.h 1197 1205 ====================

```
/*
 *   Recover an error report and clear atomically
 */

static inline int sock_error(struct sock *sk)
{
     int err=xchg(&sk->err,0);
```

```
1204     return -err;
1205 }
```

这个 inline 函数一方面将 sk->err 的内容读入变量 err 中，一方面将 sk->err 清 0。在插口操作的一些关键函数和关键路径上都安排了对 sock 结构中的这个出错代码的检测。例如，在 sock_alloc_send_skb()的 while 循环中以及在 unix_stream_recvmsg()的 do-while 循环中都安排了这样的检测。

此外，sock 结构中还有个函数指针 state_change，在 sock 结构初始化时设置成指向 sock_def_wakeup()。每当一个插口的状态改变时，就要通过这个指针调用相应的函数，以唤醒可能正在睡眠等待该插口改变状态的进程。现在，一对互相连接的插口中有一个已经关闭了，这当然引起另一方的状态改变，所以要调用这个函数（386 行）。

最后，还要通过 sk_wake_async()向所有正在对方插口上异步操作的进程发出 SIGIO 信号，让它们在对方插口上进行一次操作，从而得知该插口的状态改变。

继续往下看函数 unix_release_sock()代码的余下部分：

==================== net/unix/af_unix.c 395 429 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()]

```
395     /* Try to flush out this socket. Throw out buffers at least */
396
397     while((skb=skb_dequeue(&sk->receive_queue))!=NULL)
398     {
399          if (state==TCP_LISTEN)
400               unix_release_sock(skb->sk, 1);
401          /* passed fds are erased in the kfree_skb hook      */
402          kfree_skb(skb);
403     }
404
405     if (dentry) {
406          dput(dentry);
407          mntput(mnt);
408     }
409
410     sock_put(sk);
411
412     /* ---- Socket is dead now and most probably destroyed ---- */
413
414     /*
415      * Fixme: BSD difference: In BSD all sockets connected to use get
416      *     ECONNRESET and we die on the spot. In Linux we behave
417      *     like files and pipes do and wait for the last
418      *     dereference.
419      *
420      * Can't we simply set sock->err?
421      *
422      *     What the above comment does talk about? --ANK(980817)
423      */
424
425     if (atomic_read(&unix_tot_inflight))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
426         unix_gc();          /* Garbage collect fds */
427
428      return 0;
429  }
```

如果插口的接收队列中有报文，当然要逐个把它们摘下来释放。读者已经在前一节中看到过函数 kfree_skb()的代码。但是当插口的状态为 TCP_LISTEN 时是个特殊情况，因为此时接收队列中的报文都是连接请求。对方已经为 server 插口创建好一个 sock 结构，就等着 server 方进程通过 accept()为 server 插口创建一个新的 socket 结构，并使它们互相挂上钩。现在 server 插口既然撤销了，就要把所有这些由 client 方创建好的 sock 结构也撤销，所以这里对其递归调用 unix_release_sock()。

我们知道，通过 bind()为插口指定一个常规的、文件名形式的"地址"时，要在文件系统的目录树中创建一个节点，并使 sock 结构中的指针 protinfo.af_unix.dentry 指向内存中代表着这个目录项的 dentry 结构，同时也递增了这个结构中的使用计数。现在，如果这个 dentry 结构存在（见 405 行），就要通过 dput()递减它的使用计数，要是递减到了 0 就要将数据结构释放。对目录项所在文件系统的安装"连接件"也是一样，要通过 mntput()递减其使用计数，要是递减到了 0 就要将此文件系统拆除。

对 sock 结构本身也要递减其使用计数。同样，如果递减到了 0 就要将此结构释放。如果递减后不是 0 呢？那就说明有别的进程还在使用这个结构。这是可能的，毕竟对 sock 结构所加的锁在 373 行处就已解除了。但是，不管是谁，对此结构调用的最后一次 sock_put()总会将其释放。

最后，还有个重要的事情要做。以前讲过，可以通过 sendmsg()把对已打开文件的访问授权给其它进程。发送报文的一方要在 unix_attach_fds()中通过 unix_inflight()将这些已打开文件记下账，表示对这些已打开文件的访问授权已经在发送的过程中，即所谓" inflight"（在飞行中）。而接收方则在接收到这些授权以后要在 unix_detach_fds()中通过 unix_notinflight()"销账"。所以，当全局量 unix_tot_inflight 非 0 时，就表示有这样的授权尚在"飞行"中，还没有被其目标进程所接收。同时，当通过 sendmsg()发送对已打开文件的授权时，在 scm_send()中会通过 fget()递增相应 file 结构中的共享计数，表示从这一刻起这个已打开文件已经多了一个"用户"。在目标进程接收到这个报文前，这个所谓用户就是内核。然后，当目标进程接收到这个报文时，要通过 scm_recv()调用 scm_detach_fds()。在里面先通过 get_file()递增相应 file 结构中的共享计数，表示接收进程也成了这个已打开文件的用户，随后又通过 __scm_destroy()递减这个 file 结构中的共享计数，表示内核不再是这个已打开文件的用户了。一旦载送着文件访问授权的报文被接收方进程所接收，共享这个已打开文件的用户就都是进程了，或迟或早这些进程总会关闭这个文件，而最后关闭这个文件的进程会将共享计数减至 0 而最终真正将其"关闭"。

可是，有两种情况要特别加以考虑。

第一种情况是我们这个插口的接收队列中有一个或几个报文载送着这样的文件访问授权。显然，由于正在关闭这个插口，这些报文都要被丢弃了。但是，所谓"丢弃"绝不是简单地释放缓冲区了事，因为对载送着文件访问授权的报文还得负起递减相应 file 结构中的共享计数的责任，否则这些文件永远也不可能真正关闭。我们回过去看一下前面的 402 行，接收队列中的报文（即 sk_buff 结构）是通过 kfree_skb()释放的，我们已在前一节中看过它的代码。它最终会通过 sk_buff 结构中的函数指针 destructor 调用一个函数。这个函数一般都指向 sock_wfree()，但是当报文载送着文件访问授权时则指向 unix_destruct_fds()，这是在发送报文时由 unix_attach_fds()设置的。函数 unix_destruct_fds()的代码在 net/unix/af_unix.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== net/unix/af_unix.c 1123 1133 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>kfree_skb()>__kfree_skb()>unix_destruct_fds()]

```
static void unix_destruct_fds(struct sk_buff *skb)
{
    struct scm_cookie scm;
    memset(&scm, 0, sizeof(scm));
    unix_detach_fds(&scm, skb);

    /* Alas, it calls VFS */
    /* So fscking what? fput() had been SMP-safe since the last Summer */
    scm_destroy(&scm);
    sock_wfree(skb);
}
```

先看这里的 unix_detach_fds()，这也在 net/unix/af_unix.c 中：

==================== net/unix/af_unix.c 1111 1121 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>kfree_skb()>__kfree_skb()>unix_destruct_fds()>unix_detach_fds()]

```
static void unix_detach_fds(struct scm_cookie *scm, struct sk_buff *skb)
{
    int i;

    scm->fp = UNIXCB(skb).fp;
    skb->destructor = sock_wfree;
    UNIXCB(skb).fp = NULL;

    for (i=scm->fp->count-1; i>=0; i--)
        unix_notinflight(scm->fp->fp[i]);
}
```

这个函数的作用有三点，先是使 scm_cookie 结构中的指针 fp 指向随同 sk_buff 结构发送过来的 scm_fp_list 结构；其次将 sk_buff 结构中的函数指针 destructor 恢复成指向 sock_wfree；再就是为每个授权访问的文件通过 unix_notinflight()“销账”。但是，至此还没有触及相应 file 结构中的共享计数，那是在 scm_destroy()中完成的，有关的代码在 iscm.h 和 scm.c 中：

==================== include/net/scm.h 27 31 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>kfree_skb()>__kfree_skb()>unix_destruct_fds()>scm_destroy()]

```
static __inline__ void scm_destroy(struct scm_cookie *scm)
{
    if (scm && scm->fp)
        __scm_destroy(scm);
}
```

==================== net/core/scm.c 101 112 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>kfree_skb()>__kfree_skb()>unix_destruct_fds()>scm_destroy()>__scm_destroy()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
void __scm_destroy(struct scm_cookie *scm)
{
    struct scm_fp_list *fpl = scm->fp;
    int i;

    if (fpl) {
        scm->fp = NULL;
        for (i=fpl->count-1; i>=0; i--)
            fput(fpl->fp[i]);
        kfree(fpl);
    }
}
```

如前所述，这里的 fput()递减相应 file 结构中的共享计数。当然，如果递减后成为 0 就将此文件最终关闭。

第二种情况就更加复杂，，那就是我们刚从将要关闭的插口发送了一个报文，将对此插口的访问授权给对方，但是该报文尚在“飞行”中，还没有被对方接收，而这一边的插口倒要关闭了。下面我们通过两个情景来说明可能存在的问题。

假定有两个进程 A 和 B，还有两个 Unix 域插口 sa 和 sb；并且 A 是 sa 惟一的用户，而 B 是 sb 惟一的用户。先来看第一个情景：

(1) A 通过 sa 发送一个报文到 sb，把对 sa 的访问授权给 B，因此已经递增了 sa 的 file 结构中的共享计数。该报文已经挂入到 sb 的接收队列中，所以 A 的 sendmsg()操作已经成功返回。可是，B 没有在 sb 上接收报文，所以该报文还在 sb 的接收队列中等待被接收，对插口 sa 的访问授权还在“飞行”中。

(2) 接着，A 通过 close()关闭 sa。内核递减 sa 的 file 结构中的共享计数，但是由于在递减前的计数为 2，所以递减后并未到达 0。因此，系统调用 close()至此就完成了，并未对 sa 执行 sock_close()。可是，进程 A 却从此失去了与 sa 的联系。

(3) B 没有从 sb 接收报文，但是却关闭了 sb。由于在此之前 sb 的 file 结构中的共享计数为 1，所以递减后达到了 0，因此内核对 sb 执行 sock_close()。在执行 sock_close()的过程中，内核对 sb 接收队列中的每个报文都执行 kfree_skb()。当对上述由 A 发出的那个报文执行 kfree_skb()时，在 scm_destroy()中会对 sa 的 file 结构调用 fput()。这时候，sa 的 file 结构的共享计数就变成了 0，从而引起 sock_close()对 sa 的执行而将其最终关闭撤销了。最后，sb 插口本身也最终地关闭、撤销了。

当然，如果 B 接收了上述报文，那么 B 就成了 sa 的惟一用户。最后 B 终究要关闭 sa，那时 sa 也最终得到关闭和撤销。插口的 sa 和 sb 之间的这种关系可以推广到更多个插口而形成一条链，但是最终这条链中的所有插口都能正常地关闭和撤销。所以，这个情景没有什么问题。

再来看第二个情景：

(1) A 通过 sa 发送一个报文到 sb，把对 sa 的访问授权给 B。该报文已经挂入到 sb 的接收队列中，但尚未被接收，所以对 sa 的访问授权还在“飞行”中。

(2) B 也通过 sb 发送一个报文到 sa，把对 sb 的访问授权给 A。同样地，该报文已经挂入到 sa 的接收队列中，但尚未被接收，所以对 sb 的访问授权也还在“飞行”中。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

(3) A 通过 close()关闭 sa，但是因对 sa 的共享计数递减后未达到 0 而不能对 sa 执行 sock_close()，可是进程 A 与插口 sa 之间的联系却已经切断了。

(4) B 也通过 close()关闭 sb。同样地，因对 sb 的共享计数递减后未达到 0 而不能对 sb 执行 sock_close()，可是进程 B 与插口 sb 之间的联系也已经切断了。

如果不采取措施的话，那么 A 发出的报文就会永远留在 sb 的接收队列中，而 B 发出的报文则会永远留在 sa 的接收队列中。两个插口的共享计数都是 1，却没有一个进程能访问到这两个插口的任何一个。插口本身是不能“生活自理”的，它自己不会关闭和撤销自己。于是，这两个插口，也就是有关的数据结构，连同接收队列中的所有报文都已成了“废料”，但却要永远占着位置不放。同样，插口 sa 和 sb 之间的这种关系也可以推广到更多个插口。如果说在前一种情景所涉及的插口构成一个链的话，那么在这个情景中所构成的是一个环，而区别也正在于此。这种现象与“死锁”的表现虽然不同，原理却是相通的。

对这一类问题的对策无非是“防”与“化”两个字。所谓“防”就是防止这种现象的发生；而“化”则首先要能检测到这种现象的发生，然后就来化解。具体到上述问题，“防”是很难的，所以只好在“化”字上做文章。为此目的，内核中设计了一个专门用来解决这个问题的“废料收集”机制，具体就是由前面 unix_release_sock()代码中 426 行处调用的 unix_gc()实现的。这个函数的代码在 net/unix/garbage.c 中，我们分段阅读：

==================== net/unix/garbage.c 166 216 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_gc()]

```
/* The external entry point: unix_gc() */

void unix_gc(void)
{
	static DECLARE_MUTEX(unix_gc_sem);
	int i;
	unix_socket *s;
	struct sk_buff_head hitlist;
	struct sk_buff *skb;

	/*
	 *	Avoid a recursive GC.
	 */

	if (down_trylock(&unix_gc_sem))
		return;

	read_lock(&unix_table_lock);

	forall_unix_sockets(i, s)
	{
		s->protinfo.af_unix.gc_tree=GC_ORPHAN;
	}
	/*
	 *	Everything is now marked
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     */

    /* Invariant to be maintained:
         - everything unmarked is either:
         -- (a) on the stack, or
         -- (b) has all of its children unmarked
         - everything on the stack is always unmarked
         - nothing is ever pushed onto the stack twice, because:
         -- nothing previously unmarked is ever pushed on the stack
     */

    /*
     *   Push root set
     */

    forall_unix_sockets(i, s)
    {
         /*
          *   If all instances of the descriptor are not
          *   in flight we are in use.
          */
         if(s->socket && s->socket->file &&
            file_count(s->socket->file) > atomic_read(&s->protinfo.af_unix.inflight))
              maybe_unmark_and_push(s);
    }

```

“废料”收集的过程是不容许两个进程同时进行的，所以要通过内核信号量 unix_gc_sem 把它保护起来。另一方面，在这个过程中要扫描和处理杂凑表 unix_socket_table[]中的所有队列，所以在此期间也不容许为任何其它目的来变动这些队列，因此还要将整个 unix_socket_table[]加上锁。代码中的 forall_unix_socket()是个宏定义，其定义在 include/net/af_unix.h 中：

==================== include/net/af_unix.h 16 18 ====================

```

#define forall_unix_sockets(i, s) for (i=0; i<=UNIX_HASH_SIZE; i++) \
                              for (s=unix_socket_table[i]; s; s=s->next)
```

显然，这是针对杂凑表中所有的队列，扫描队列中的所有 sock 结构。那么，扫描这些 sock 结构干什么呢？第一趟扫描时把所有的 sock 结构都标志成 GC_ORPHAN。也就是说，先假定所有的插口都已经成了前述“生活不能自理”的“孤儿”。Unix 域插口的 sock 结构中专为废料收集而设置了一个指针 profile.af_unix.gc_tree。只有 Unix 域插口才支持对已打开文件的授权，所以只有 Unix 域的插口才有废料收集的问题。常数 GC_ORPHAN 的定义也在 net/unix/garbage.c 中：

==================== net/unix/garbage.c 85 92 ====================

```
/* Internal data structures and random procedures: */

#define GC_HEAD        ((unix_socket *)(-1))
#define GC_ORPHAN  ((unix_socket *)(-3))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static unix_socket *gc_current=GC_HEAD;   /* stack of objects to mark */

atomic_t unix_tot_inflight = ATOMIC_INIT(0);
```

这里我们同时列出了其它几个定义。其中 unix_tot_inflight 就是当前正在“飞行”中的已打开文件访问授权的个数。向指针 go_current 则在废料收集的过程中用来构筑一个“后进先出”队列。

顺便提一下，我们是在函数 unix_release_sock()的末尾处调用 unix_gc()的。此时我们要关闭并撤销的插口本身已经与杂凑表无关，它的 sock 结构在刚进入 unix_release_sock()时就已经通过 unix_remove_socket()从杂凑表的队列中摘除了。事实上，“废料收集”与正欲撤销的插口本身无关，只不过是乘机让它承担一点义务，作出一点贡献而已。

第一趟扫描将所有的插口都假定为“孤儿”，当然不见得就符合事实，所以第二趟扫描就要来加以甄别了。根据什么准则来甄别呢？那就是比较插口的 file 结构中的共享计数和 sock 结构中的另一个计数器 protinfo.af_unix.inlight。我们以前看到过，当将对一个插口的访问授权通过 sendmsg()发送给另一个进程时，要调用 unix_inflight()递增该插口的 sock 结构中的这个计数器，以及全局性的计数器 unix_tot_inflight，而在对方接收了这个报文时则要调用 unix_notinflight()递减这两个计数器。当然，对插口的 file 结构中的共享计数也要作类似的处理。如果我们比较这两个计数器的大小，则可以得到如下的结论：

(1) 文件共享计数 > 授权报文计数：正常。说明除正在“飞行”中的授权报文外至少还有一个用户，而这个用户必然是一个进程。所以，这个插口不是“孤儿”。

(2) 文件共享计数 = 授权报文计数：不正常。这个插口已经没有真正意义上的“用户”了，所以可能是“孤儿”。

(3) 文件共享计数 < 授权报文计数：不可能发生。

在第二趟扫描中，凡遇正常的 sock 结构就通过其指针 protinfo.af_unix.gc_tree 将其链入到队列 gc_current 的前部，或者说将其推入“堆栈”gc_current。实现这一操作的函数代码如下（见 net/unix/garbage.c）：

==================== net/unix/garbage.c 156 163 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_gc()>maybe_unmark_and_push()]

```
extern inline void maybe_unmark_and_push(unix_socket *x)
{
    if (x->protinfo.af_unix.gc_tree != GC_ORPHAN)
        return;
    sock_hold(x);
    x->protinfo.af_unix.gc_tree = gc_current;
    gc_current = x;
}
```

注意，将一个 sock 结构链入到 gc_current 队列中，并不意味着这个结构就脱离了杂凑表 unix_socket_table[]的队列，sock 结构可以通过不同的指针链入不同的队列。

这样，第二趟扫描以后，凡是其指针 protinfo.af_unix.gc_tree 仍为 GC_ORPHAN 的插口很可能就是“孤儿”了。但是，至此还不能肯定所有这些插口都已经成了“孤儿”，因为这实际上取决于授权的对

PDF-XCHANGE Click to buy NOW! www.docu-track.com

象，更确切地说是接收授权报文的插口。以上面所举例子中的插口 sa 为例。对 sa 的访问授权有可能已经通过多个报文发送到了多个插口，然后 A 关闭了 sa。只要有一个对访问 sa 的授权报文是在一个“正常”插口的接收队列中，那么这个报文就有希望被某个进程接收，从而使 sa 又有了真正意义上的用户，那个进程自会负起最后关闭并撤销 sa 的责任。所以，还要进一步甄别，这一次是检查所有“正常”插口的接收队列中的每一个报文，而所有“正常”插口的 sock 结构都已经在队列 gc_current 中了。

继续看 unix_gc 的代码：

==================== net/unix/garbage.c 217 266 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_gc()]

```
	/*
	 *	Mark phase 
	 */

	while (!empty_stack())
	{
		unix_socket *x = pop_stack();
		unix_socket *sk;

		spin_lock(&x->receive_queue.lock);
		skb=skb_peek(&x->receive_queue);
		
		/*
		 *	Loop through all but first born 
		 */
		
		while(skb && skb != (struct sk_buff *)&x->receive_queue)
		{
			/*
			 *	Do we have file descriptors ?
			 */
			if(UNIXCB(skb).fp)
			{
				/*
				 *	Process the descriptors of this socket
				 */
				int nfd=UNIXCB(skb).fp->count;
				struct file **fp = UNIXCB(skb).fp->fp;
				while(nfd--)
				{
					/*
					 *	Get the socket the fd matches if
					 *	it indeed does so
					 */
					if((sk=unix_get_socket(*fp++))!=NULL)
					{
						maybe_unmark_and_push(sk);
					}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
255                     }
256                 }
257                 /* We have to scan not-yet-accepted ones too */
258                 if (x->state == TCP_LISTEN) {
259                     maybe_unmark_and_push(skb->sk);
260                 }
261                 skb=skb->next;
262             }
263             spin_unlock(&x->receive_queue.lock);
264             sock_put(x);
265         }
266
```

我们把这一段代码留给读者自己阅读。这里的 pop_stack()从队列 gc_current 的前部摘下一个 sock 结构（所以是“后进先出”），而 empty_stack()则测试该队列的长度，函数 skb_peek()从一个接收队列中取第一个报文（即 sk_buff 结构）的指针，但并不将其摘下。

此处给出 pop_stack()等的代码如下（net/unix/garbage.c）：

==================== net/unix/garbage.c 140 154 ====================

```
140  /*
141   *   Garbage Collector Support Functions
142   */
143
144  extern inline unix_socket *pop_stack(void)
145  {
146      unix_socket *p=gc_current;
147      gc_current = p->protinfo.af_unix.gc_tree;
148      return p;
149  }
150
151  extern inline int empty_stack(void)
152  {
153      return gc_current == GC_HEAD;
154  }
```

注意，maybe_unmark_and_push()对于已经进入 gc_current 队列的 sock 结构不起作用（见 159 行）。

经过这两遍甄别以后，所有仍是 GC_ORPHAN 的插口就最后定性为“孤儿”了。对这些“孤儿”怎么办呢？扫描其接收队列中的所有报文，凡发现载有访问授权的报文就将其从接收队列中摘下来集中到一个临时的队列 hitlist 中，最后对这些报文执行 kfree_skb()，其代码我们已在前面看到过。注意这里只要对载有访问授权的报文调用 kfree_skb()就行了，对于已定性为“孤儿”的 sock 结构并不需要特地加以处理，因为这一定已经隐含在对某个报文的 kfree_skb()中了。至于“孤儿”sock 结构的接收队列中的其它报文，也自会得到释放（请读者想一下，为什么？）

最后给出 unix_gc 代码的余下部分：

==================== net/unix/garbage.c 267 304 ====================

[sys_close()>filp_close()>fput()>sock_close()>sock_release()>unix_release()>unix_release_sock()>unix_gc()]

```
267      skb_queue_head_init(&hitlist);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

        forall_unix_sockets(i, s)
        {
                if (s->protinfo.af_unix.gc_tree == GC_ORPHAN)
                {
                        struct sk_buff *nextsk;
                        spin_lock(&s->receive_queue.lock);
                        skb=skb_peek(&s->receive_queue);
                        while(skb && skb != (struct sk_buff *)&s->receive_queue)
                        {
                                nextsk=skb->next;
                                /*
                                 *   Do we have file descriptors ?
                                 */
                                if(UNIXCB(skb).fp)
                                {
                                        __skb_unlink(skb, skb->list);
                                        __skb_queue_tail(&hitlist,skb);
                                }
                                skb=nextsk;
                        }
                        spin_unlock(&s->receive_queue.lock);
                }
                s->protinfo.af_unix.gc_tree = GC_ORPHAN;
        }
        read_unlock(&unix_table_lock);

        /*
         *   Here we are. Hitlist is filled. Die.
         */

        while ((skb=__skb_dequeue(&hitlist))!=NULL) {
                kfree_skb(skb);
        }

        up(&unix_gc_sem);
}
```

如前所述，对 unix_gc()的调用其实并不是非得放在 unix_release_sock()中。例如，完全可以周期性地调用这个函数。但是，放在 unix_release_sock()的末尾还是比较合适的。也许读者会问，为什么不把对这个问题的处理放在系统调用 close()的开头呢？例如，在递减了 file 结构中的共享计数以后，如果没有达到 0 就与 sock 结构中的授权计数比较一下，这样马上就可以知道是否有本插口发生的授权报文正在“飞行”中，这样不是更有效吗？问题在于，文件系统操作界面的函数都是通用的，必须适用于所有不同类型的文件，而插口只是其中的一种。以关闭文件为例，内核中由 sys_close()调用 filp_close()，再由 filp_close()调用 fput()，所有这些函数都只是在 vfs 层对抽象的文件进行操作，根本就不知道所操作的到底是个什么样的文件，所以显然是不合适的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

7.9 其它

如前所述，“插口”机制有两个用户程序界面，一个是为插口专设的，另一个就是通用的文件操作界面。这样，即使只考虑 Unix 域的插口，内核中与之有关的函数就自然更多一些了。我们已经把主要的和比较复杂的 Unix 域的插口操作及其代码作了详细介绍，但是限于篇幅我们只好把剩下的一些代码留给读者自己阅读了。

从插口操作界面，也就是系统调用 socketcall()在内核中的入口 sys_socketcall()的角度来看，留下来的有这么一些：sys_getsockname()、sys_getpeername()、sys_socketpair()、sys_shutdown()、sys_getsockopt()以及 sys_setsockopt()。这些函数也要通过数据结构 unix_dgram_ops 或 unix_stream_ops 跳转到 unix 域插口对这些操作的实现，分别为 unix_getname()、unix_socketpair()、unix_shutdown()。函数 sys_getpeername()与 sys_getsockname()的不同之处在于后者是取插口本身的地址，而前者是取对方插口的地址（限于已调用 connect()以后）。所以最终都是由 sys_getname()完成的。另外，sys_getsockopt()和 sys_setsockopt()二者都是为网络环境设计的，因为在具体的网络通信规程中常常有许多可选项要设置，而在 unix 域中则不存在这个问题。所以，在 unix_dgram_ops 和 unix_stream_ops 两个结构中，有关的函数指针为 sock_no_getsockopt()和 sock_no_setsockopt()，这些函数都只是返回一个出错代码-EOPNOTSUPP，表示系统不支持这些操作。

总的来说，剩下的这些函数都是很简单、很直截了当的。例如，unix_socketpair()的代码为（见 net/unix/af_unix.c）：

==================== net/unix/af_unix.c 1015 1036 ====================

[sys_socketcall()>sys_socketpair()>unix_socketpair()]

```
static int unix_socketpair(struct socket *socka, struct socket *sockb)
{
	struct sock *ska=socka->sk, *skb = sockb->sk;

	/* Join our sockets back to back */
	sock_hold(ska);
	sock_hold(skb);
	unix_peer(ska)=skb;
	unix_peer(skb)=ska;
	ska->peercred.pid = skb->peercred.pid = current->pid;
	ska->peercred.uid = skb->peercred.uid = current->euid;
	ska->peercred.gid = skb->peercred.gid = current->egid;

	if (ska->type != SOCK_DGRAM)
	{
		ska->state=TCP_ESTABLISHED;
		skb->state=TCP_ESTABLISHED;
		socka->state=SS_CONNECTED;
		sockb->state=SS_CONNECTED;
	}
	return 0;
}
```

参数 socka 和 sockb 为已经在 sys_socketpair()中通过 sock_create()创建好的两个 socket 数据结构。这里的 1022 行和 1023 行使它们的 sock 结构各自指向对方，如果是“有连接”插口则还要设置好各自的有限状态以及插口的状态。

另一方面，从文件系统的操作界面来看，则还有这么几个函数：sock_lseek()、sock_poll()、sock_ioctl()、sock_mmap()以及 sock_fasync()。其中 sock_lseek()只是返回一个出错代码，因为插口并不支持 lseek()。函数 sock_poll()用于系统调用 select()；sock_ioctl()用于系统调用 ioctl()。相应的 unix 域函数为 unix_poll()、datagram_poll()以及 unix_ioctl()。读者可以结合“文件系统”和“字符设备驱动”这两章把它们读懂。操作 sock_mmap()在 Unix 域中的实现为 sock_no_mmap()，就是说 Unix 域插口并不支持 mmap()。最后，sock_fasync()用来让插口在每当有报文（包括连接请求）到达时就向若干进程发出 IOSIG 信号，从而实现对报文的异步接收。这个函数并没有特别针对 Unix 域插口的底层操作，而是通用于所有插口。我们在讲述 sock_close()时已经列出了 sock_close()的代码，但是只讲了当调用参数 on 为 0 时，也就是要停止异步接收时的那部分代码。不过，读懂了一个方向的操作（停止异步接收）的代码以后，要读懂其相反方向（启动异步接收）的代码应该是比较容易的了。

最后，我们建议读者把 sock 数据结构（以及 sk_buff）的定义打印出来，再根据我们讲到的内容，试着为数据结构中的每个字段（成分）（只要是已经讲到过的）的用途和作用加上注释。如果你觉得很多字段的用途和作用实在不是三言两语能够讲清的，你就明白了为什么我们没有在一开始就列举该数据结构各个字段的作用了。在这种情况下，你不妨在打印的清单上写上“见×××页”等字样，以备日后查阅。

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

第8章 设备驱动

8.1 概述

设备驱动在系统中的重要地位是无需多加说明的，计算机最基本的三个物质基础就是 CPU、内存以及输入 / 输出（I/O）设备。严格地说，离开了对设备的操作，即输入 / 输出操作，计算机本身也就失去了意义。相比之下，文件系统的存在只不过是使对设备的操作更为方便、更为有效、更有组织、更接近人类的思维方式而已。所以，文件操作是对设备操作的组织与抽象，而设备操作则是对文件操作的最终实现。

Unix 操作系统从一开始就将所有的设备（而不仅是磁盘上的文件）全都看成文件，都纳入文件系统的范畴，都通过文件操作的界面进行操作。这就意味着：

- 每一项设备都至少由文件系统中的一个文件（更确切地说是节点）代表，因而都有一个“文件名”。每个这样的“设备文件”都惟一地确定了系统中的一项设备。应用程序通过设备的文件名寻访具体的设备，而设备则像普通文件一样受到文件系统访问权限控制机制的保护。
- 应用程序通常可以通过系统调用 open()“打开”设备文件，建立起与目标设备的连接，或曰“上下文”。代表着该设备的文件节点中记载着建立这种连接所需的信息。对于执行该应用程序的进程而言，建立起的连接就表现为一个已打开文件。
- 打开了代表着目标设备的文件，即建立起与设备的连接以后，就可以通过 read()、write()、ioctl()等常规的文件操作对目标设备进行操作。从应用程序的角度看，设备文件逻辑上的空间是个线性空间。从这个逻辑空间到具体设备的物理空间的映射则由内核提供，并划分成文件操作与设备驱动两个层次。

这样，对于一个具体的设备来说，文件操作和设备驱动就成为同一事物的不同层次，而不是互相独立或平行的两个概念。从这种观点和结构模型出发，一般而言，至少可以在概念上把一个系统划分成应用、文件系统以及设备驱动这么三个层次。这不仅适用于 Unix，也可以运用到其它的操作系统，尽管具体的划分和安排可以不同。例如，通常文件系统层和设备驱动层都在内核中，但是也有的系统把文件系统放在内核外面作为应用层的一个进程。表面上看来（从进程的角度）此时文件系统与应用程序互相平行，但是如果考察对一具体文件的操作过程，就可以看出用来实现文件系统的进程实际上成了应用程序与内核之间的一个附加的层次。有些系统只提供设备驱动而不提供文件系统，由应用程序在进程内部实现其自己的文件系统（例如通过一些库函数调用）。显然这只是文件系统层的物理位置不同，而并不改变整个结构模型的层次结构。即使有些特殊的应用程序根本不使用文件系统，而直接对设备驱动层操作，那也可以认为只是一种特例，即此时的文件系统是完全“透明”的。还有些操作系统，特别是早期的操作系统，把设备驱动做成与文件系统平行，有独立的命名空间以及一套独立的操作（系统调用），让应用层直接与设备驱动打交道。表面上看来这是完全不同的结构模型，但是如果作更深入的思考就可以领悟到：实际上这只不过是实现方法上的不同，只不过是为设备驱动保留了更

多的特殊性，对设备少作了某些抽象而已，而采用一套独立的、专用的操作则有损其灵活性。Unix 的设计者正是认识到了这一点，而在 Unix 中首先明确地把设备驱动纳入了文件系统。所以，许多系统的外观也许不同，但是如果仔细加以考察就可以发现实际上说到底还是同一个结构模型。而 Unix（和 Linux）的文件系统，则是这种结构模型的典型实现，也许可以说是最自然的实现。本书上册第 5 章“文件系统”的文件系统结构图（图 5.1）就反映了这种系统结构，其文件系统层和设备驱动层都在内核中。从这个意义上说，本书第 5 章也许不应该叫“文件系统”，而应更确切地称为“磁盘文件系统”或“普通文件系统”。所以说，“文件系统”这个词的含义是很模糊的。

但是，话虽如此，对于不同的设备其文件系统层的“厚度”却有所不同。对于像磁盘这样结构性很强，并且其内容需要进一步加以组织和抽象的设备来说，其文件系统很“厚”很“重”，这一点读者在阅读“文件系统”一章时想必已有所体会。磁盘设备的复杂性来自两个方面。一方面是介质本身的结构，如“磁道”、“柱面”、“扇区”以及抽象意义上的“记录块”；另一方面是在“记录块”基础上的又一层组织和抽象，即“磁盘文件”。这样，在物理介质上的第一层抽象使操作者不必关心读 / 写的物理位置究竟在哪一个磁道，哪一个扇区；而第二层抽象则使操作者不必关心读 / 写的内容在哪一个逻辑“记录块”中。很自然地，我们把第一层抽象归入设备驱动层，而第二层抽象则归入文件系统层。但是，还有一些设备，如字符终端、字符打印机等，则由于本身并没有什么结构，甚至不存在存储介质，因而简单得多。对于这样的设备，其文件系统层自然就比较“薄”，甚至近乎“透明”。不光是文件系统层比较薄，连设备驱动层也可能很简单。另一方面，即使是对于磁盘设备，在有些应用中也可以绕过第二层抽象而简单地把它当成记录块“数组”，也就是忽略由这些记录块的内容形成的关连和组织，使原来很厚的文件系统层变得很薄。在这种情况下，我们称之为“原始”（raw）设备。当然，物理上还是同一个设备，只不过是“横看成岭侧成峰”——从不同的角度看问题而已。在这方面，将设备驱动纳入文件系统又表现出其优越性，因为对同一设备的不同驱动方式可以由不同的文件来代表。下页的示意图（图 8.1）或许将有助于读者的理解。

在这个示意图中，处于应用层中的进程通过“打开文件号”fd 与已打开文件的 file 结构相联系，每个 file 结构代表着对一个已打开文件操作的上下文。通过各个上下文，进程使用各个文件的线性逻辑空间进行文件操作。对于普通文件，即“磁盘文件”，文件的逻辑空间在文件系统层内按具体文件系统的结构和规则映射到设备的线性逻辑空间，然后在设备驱动层进一步从设备的逻辑空间映射到其物理空间。这样，一共经历了两层映射。或者，也可以反过来说，磁盘设备的物理空间经过两层抽象而成为普通文件的线性逻辑空间。而对于“设备文件”，则文件的逻辑空间通常直接就等价于设备的逻辑空间，所以在文件系统层就不需要有映射。但是，也有些设备需要在文件系统层中有一些简单的映射。从图中还可以看出，对同一个设备也可以通过不同的文件以不同的方式来操作。这里还要指出，代表着设备文件的节点都要通过某个索引节点才能寻访，而对索引节点又要通过一些目录节点才能寻访，这些目录节点实质上相当于普通文件，所以在打开设备文件的过程中即隐含着对普通文件的操作。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

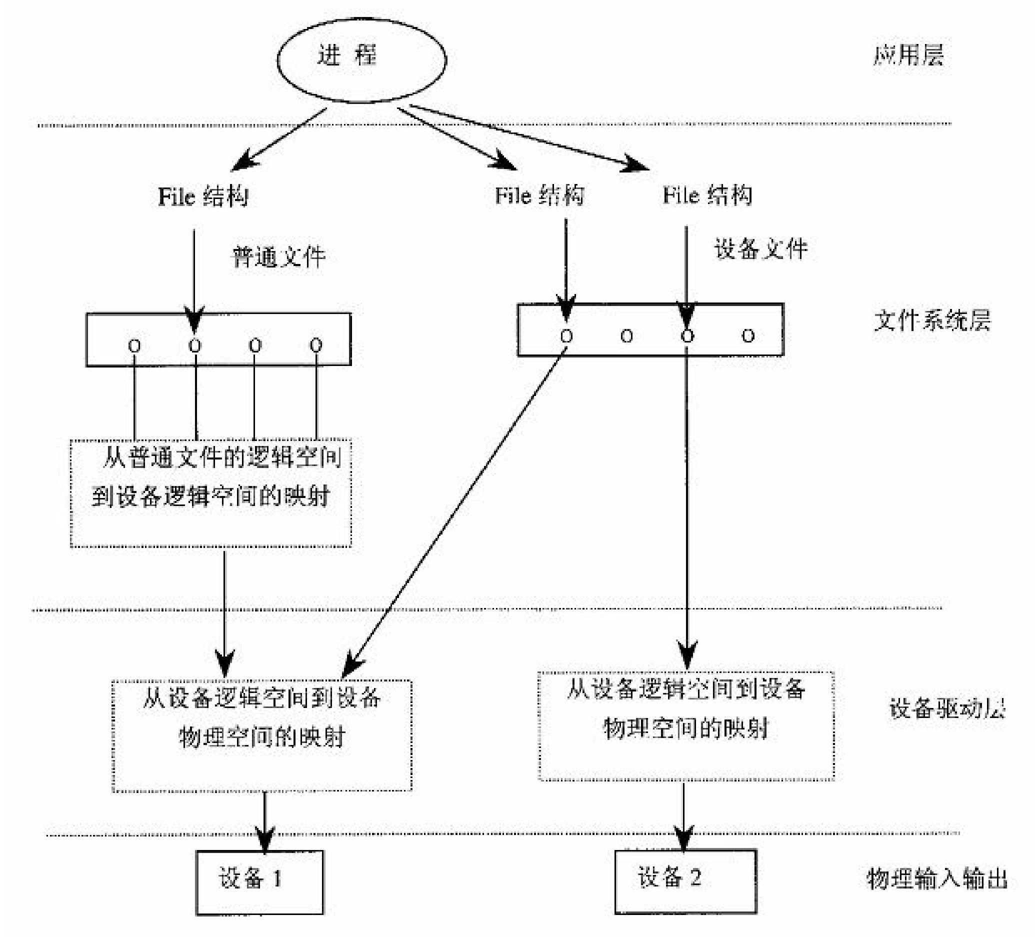


图 8.1　设备驱动分层结构示意图

Unix（以及 Linux）将设备分成两大类。一类是像磁盘那样以记录块或“扇区”为单位，成块进行输入 / 输出的设备，称为“块设备”；另一类是像键盘那样以字符（字节）为单位，逐个进行输入 / 输出的设备，称为“字符设备”。文件系统通常都建立在块设备上，但是也并无规定说在字符设备上就不可以建立文件系统。通过数十年的发展，块设备和字符设备之间的界线已经模糊了，但还是沿用着这样的划分。两类设备之间的界线之所以变得模糊，一方面是由于传统字符设备变得愈来愈复杂了，另一方面是出现了一些既像是块设备又像是字符设备的新设备。例如，网络接口卡就是这样的设备，说它是字符设备吧，它的输入 / 输出却是有结构的、成块的（报文、包、帧）；说它是块设备吧，它的“块”又不是固定大小的，大到数百甚至数千字节，小到几个字节。块设备与字符设备还有个区别，那就是块设备的介质必须是存储介质，并且支持“随机访问”（对指定地址的访问），因而系统调用 lseek()对于块设备是不可缺少的操作；而字符设备的介质则一般都是传输介质，一般只支持“顺序访问”，因而 lseek()对于字符设备意义不大。从这个意义上讲，网络设备当然就是字符设备了。不过这也不是绝对的。举例来说，在 Unix 系统中块设备都有对应的“原始设备”。例如，有块设备/dev/hda，则必有另一设备/dev/rhda，这里的字符“r”表示“raw”即“原始”的意思。原始设备都是作为字符设备对待的，虽然

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

它们的介质仍为存储介质，并且也支持随机访问。(在 Linux 中，不再为块设备提供原始设备文件，因为块设备在安装之前实际上就相当于原始状态)。所以，一般都只是把成块输入 / 输出并且打算在上面建立普通文件系统的设备才称为“块设备”。

对设备的这种类别划分并不是一个理论问题，而是个技术问题。为什么呢？前面讲过，在代表着设备的文件节点中记载着与特定设备建立连接所需的信息。这种信息由三部分构成：第一部分是文件（包括设备）的类型，第二部分是一个“主设备号”，第三部分是一个“次设备号”。其中设备类型和主设备号结合在一起惟一地确定了设备的驱动程序及其界面，而次设备号则说明目标设备是同类设备中的第几个。例如，当主设备号为 2 时，若设备类型为块设备就是指软盘驱动器，而若为字符设备则是指所谓“伪终端”(Pseudo TTY) 设备。这样的安排从 Unix 的早期一直延用至今，出于兼容的考虑又不宜轻易变动，所以每项设备都得非此即彼，或划入块设备，或划入字符设备，尽管有些设备确实非驴非马。

从早期 Unix 以来一直沿用 8 位的主设备号，这就把块设备和字符设备的种类都限制到了 256 种，现在已经感到不够用了。就字符设备而言，在 Linux 源代码的 Documentation/devices.txt 中已经将主设备号分配到了 194 号。另一方面，从早期 Unix 开始就把所有的设备文件（节点）都放在/dev 目录中，使这个目录成为一个“平面”的，而不是树状的有层次的目录。当目录中的节点数量很多时，就会使打开这些文件时的效率受到影响。而且，在风格上也与文件系统中其它的部分不一致。所以，人们提出了改进的方案，就是把设备文件都放在一个树状的特殊文件系统 devfs 中。当然，这种新的方案比原来的要好，而且在现有文件系统的框架中也很容易实现。但是，新的方案必须与原有的兼容，使已有的应用软件可以继续运行。与 devfs 有关的代码都在 fs/devfs 中，我们也专门写了一节加以介绍。不过，我们有时候还会以原来平面结构的/dev 作为实例。这是因为目前老的方案还占着主导地位，机理也比较简单，再说新的 devfs 也与之兼容。读者可以在理解了基于主 / 次设备号的设备驱动以后再结合 devfs 一节加深理解。

要使一项设备在系统中成为可见，成为应用程序可以访问的设备，首先当然是要在文件系统中有一个代表此项设备的文件节点，这是通过系统调用 mknod()实现的。可是，更重要的是在设备驱动层中要有这种设备的驱动程序。在早期的操作系统中（不仅是 Unix），设备驱动程序都静态地连接在内核中。可是，把所有可能的设备驱动程序都连接到内核中显然是不现实的，因为那样会使内核的体积大到不合理、甚至根本无法运行的程度。所以，通常都为最终用广提供一个“系统生成”的工具和手段，让用户根据具体的需要挑选一些已经事先编译好的模块，同时由系统生成工具对以高级语言编写的“设备表”加以修改，在表中加入相应驱动程序的函数指针，然后加以编译并将所挑选的模块与内核连接在一起。

这种方式给用户带来了很多不便，因为每当要往系统中增加或更换一种设备时就必须再进行一次系统生成。同时，随着网络技术的发展，不断需要将新出现的网络规程以设备驱动的形式在内核中实现，而静态连接的设备驱动程序同样为内核在这方面的功能更新和升级带来不便。Unix 在这方面的不足在一个时期内阻碍了它的进一步普及。相比之下，曾经在世界范围内得到广泛采用的 DOS 操作系统虽然简陋，但是却提供了一种手段称为 TSR（Terminate and Stay Resident），让用户程序可以动态地（在系统运行时）把一段可执行程序“粘贴”到操作系统上，并提供了系统调用让用户程序可以改变中断向量。当然，TSR 给系统的安全性带来了极大的问题，而且对于像 Unix 这种多用户、多进程的系统它根本就不现实。但是不可否认这对于 DOS 以及 Windows（直到 Windows 3.1）的普及，乃至 PC 的普

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

及起了相当大的作用。因为正是由于这样，许多 PC 兼容机和外部设备的厂家才能随时向用户提供更加新型的硬件，同时提供一张软盘让用户动态地“安装”新的驱动程序。

Unix 很早就支持所谓“共享库程序”，或称动态连接程序库，这在一定程度上缓解了这个问题，可是并不能从根本上解决问题，因为动态连接的库程序是在用户空间运行的（DOS 根本就不分用户空间和系统空间）。确实，对设备的许多操作可以在用户空间中实现，而只把物理的输入和输出留给内核。但是那样一来，设备驱动程序和设备之间的结合就不紧密，从而使效率降低甚至根本不能运转。另一方面，对于像 Unix（以及 Linux）这样的系统，当然不能照搬 TSR 一类的机制（Windows 95/98，Windows NT 都已不再使用 TSR）。设计出一种既能在多用户、多进程环境下动态地安装设备驱动程序，又能保持内核安全的机制，就成为对设计人员的一种挑战。

可安装模块正是在这样一种背景下产生的。可安装模块（module）是经过编译但尚未连接的目标代码（.o）文件，可以在系统运行时动态地“安装”到内核中。这种动态安装既可以由特权用户进行，也可以由内核在有需要时自动地启动（设想在网络环境下接收到了一个特殊的报文，而实现这种报文规程的模块尚未装入）。在内核的源代码中可以规定允许“移出”（export）内核的一些符号，如子程序入口、全局变量等，使可安装模块可以在程序中调用这些内核子程序或访问这些全局变量。对这些符号的引用是在模块安装的时候连接解决的，所以安装的过程实际上就包括了连接的过程。由于连接的部分对象已经装入内存并且已在运行，所以是动态连接。例如，我们在第 3 章中讲到内核中有个全局量 jiffies，这个变量在每次时钟中断时都要递增，因此可以用作基本的计时手段。假定在一个可安装模块中需要访问这个变量，就可以在其源代码中把它说明为“外部”（extern）变量。在编译时，gcc 知道这是一个需要在连接时解决的外部符号，但是并不知道这个变量在哪里，所以就将其地址暂时空着，留待连接时再来填写。到安装模块时，应用程序可以通过系统调用向内核查询变量 jiffies 所在的位置，只要这个变量是允许移出的，内核就会将其地址返回给应用程序。而应用程序则将其地址填入该模块中所有访问这个变量的指令内以及其它所有要用到其地址的地方。这样，变量 jiffies 就“连接”上了。

这些符号连接操作是由一个实用程序/sbin/insmod 完成的。这个实用程序不但负责模块与内核的连接，也负责把模块的目标文件（.o 文件）“装入”到内核空间。所以，insmod 的功能与 ld 相似，但是 insmod 所进行的是动态的（运行时的）连接和装入，而 ld 所进行的则是静态的连接和装入。与/sbin/insmod 相对应，还有一个应用程序/sbin/rmmod，其作用是将一个已安装的模块从内核中拆除。当然，像 insmod 一样，只有特权用户才能执行 rmmod。

除这两个实用程序外，Linux 为此而增设了若干系统调用，可安装模块机制主要就是通过这些系统调用实现的。在本章中，我们将与读者一起阅读这些系统调用的代码。

在这里，我们不妨将可安装模块与 TSR 程序作一简单的比较（尽管这二者实际上并没有多大的可比性）。在 TSR 程序中，是无法与内核中的符号（子程序、全局量）作符号连接的，而只能根据一些约定（将这些变量放在指定的地址上）来访问一些重要的全局量，以及通过陷阱指令调用由 BIOS 提供的功能和操作。例如 INT 9 为显示器屏幕操作，INT 13 为磁盘操作，还有就是通过 INT 21 进行的 DOS 系统调用，等等。这样，一方面内核中的很多资源（除系统调用和一些约定的变量地址外）并未向 TSR 程序开放，而另一方面则又完全不设防，一个 TSR 程序完全可以为所欲为。例如，在某个条件满足时突然通过 INT 13 将整个磁盘上的数据全部写成 0。光凭这一点，一些病毒的肆虐和恶劣表现就不足为奇了。而在 Linux 的可安装模块中，由于 Linux 在运行时与 BIOS 没有关系（仅在引导时用到 BIOS），一方面凡是由内核“移出”的所有符号都可以在模块中引用，从而扩大了内核向可安装模块开放的资

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

源范围；另一方面，除这些特意移出的符号以及系统调用以外，应用程序就别无途径可以直接访问内核中的资源。而系统调用，特别是用于文件系统操作的系统调用，是受到访问权限机制的严格控制的。这样，向可安装模块开放的子程序和全局量都经过仔细的权衡，即使不能保证这些资源全都是“无毒”的，也还有最后一道防线，因为只有特权用户才能安装这些模块。而且，由于 Linux 内核的源代码是公开的，有特殊要求的用户还可以对移出符号的名单加以调整。例如，要是知道不需要安装有关磁盘的模块，就可以将 ll_rw_block()等有关块设备驱动的函数从内核的“移出名单”中剔除。目前，可安装模块机制是作为内核的一个可选项提供的，用户可以在编译内核时通过 CONFIG_MODULES 选择包含或排除对这种机制的支持。

设备驱动层是直接与物理设备打交道的，在实际的实现中则因系统的结构和具体设备的物理特性而有不同的驱动方式。事实上，多数设备都是“中断驱动”的，而块设备往往都采用 DMA（“直接访问内存”）方式，所以物理设备的输入 / 输出从本质上说大多是异步的。相比之下，文件操作既可以是同步的，也可以是异步的，但是多数情况下是同步的。以从键盘读一个字符的过程为例，用户进程通过系统调用 read()企图从标准输入文件读一个字符，但是真正的、物理意义上的从键盘读入通常并不是发生在用户进程调用 read()的瞬间。如果在此之前用户已经按了键，那么所按的字符已经通过中断服务程序读了进来，放在缓冲区中等待由进程读取，此时上述 read()操作立刻就可以完成而返回这个字符。但是，如果缓冲区中没有字符可读，那当前进程通常就要睡眠等待。等待到什么时候呢？等待到用户按键的时候，那是异步的，也就是无法预测何时会发生的。从这个意义上说，设备驱动程序是上层的同步操作与底层的异步操作之间的桥梁。

设备驱动程序要直接访问外部设备或其接口卡上的物理电路，这部分电路通常都是以“寄存器”形式出现的。根据访问外部设备寄存器的不同方式，可以把 CPU 分成两大类。一类 CPU 把这些寄存器看作内存的一部分，寄存器参与内存统一编址，访问寄存器就通过一般的访问内存指令进行，所以这种 CPU 没有专门用于设备 I/O 的指令。这样的 CPU 有 M68K、Power PC 等等。另一类 CPU 将外部设备的寄存器看成一个独立的地址空间，所以访问内存指令不能用来访问这些寄存器，而要为对外部设备寄存器的读 / 写单独设置专用的指令，如 in、out 等。i386 就是属于这一类的 CPU。可想而知，跟这些寄存器直接有关的代码是因处理器而异的。

就 i386 结构的 CPU 而言，用于外部设备寄存器读 / 写的指令主要就是两条，即 in 和 out。但是，像访问内存指令一样，根据读 / 写的对象为字节、字或长字而有 inb/inw/inl 和 outb/outw/outl 等变形。不过，Linux 内核的代码中一般都不直接使用这些指令，而将这些函数“包装”在一些相应的函数中，这些函数有 inb()、outb()、inw()、outw()等。

读者在第 3 章中曾经看到，内核中有些函数是利用 C 语言的编译时字符串拼接功能、由 gcc 在预处理阶段生成出来的，所以如果通过字符串搜索工具（如 grep）在源代码中搜索这些函数的定义就会找不到。这里，inb()、outb()这些函数的定义也是这样，是由 gcc 在编译时的预处理阶段根据一些宏定义生成出来的。下面我们就来看这些函数的生成，有关的代码都在 include/asm-i386/io.h 中。

先看 outb()/outw()/outl()的生成：

==================== include/asm-i386/io.h 92 94 ====================

```
__OUT(b,"b",char)
__OUT(w,"w",short)
__OUT(l,,int)
```

这里的__OUT()是宏定义，其代码为：

==================== include/asm-i386/io.h 58 60 ====================
```
58  #define __OUT(s,s1,x) \
59  __OUT1(s,x) __OUT2(s,s1,"w") : : "a" (value), "Nd" (port)); } \
60  __OUT1(s##_p,x) __OUT2(s,s1,"w") __FULL_SLOW_DOWN_IO : : "a" (value), "Nd" (port));} \
```

再看宏定义__OUT1、__OUT2 以及__FULL_SLOW_DOWN_IO：

==================== include/asm-i386/io.h 52 56 ====================
```
52  #define __OUT1(s,x) \
53  extern inline void out##s(unsigned x value, unsigned short port) {
54
55  #define __OUT2(s,s1,s2) \
56  __asm__ __volatile__ ("out" #s " %" s1 "0,%" s2 "1"
```
==================== include/asm-i386/io.h 46 46 ====================
```
46  #define __FULL_SLOW_DOWN_IO __SLOW_DOWN_IO
```
==================== include/asm-i386/io.h 38 38 ====================
```
38  #define __SLOW_DOWN_IO "\njmp 1f\n1:\tjmp 1f\n1:"
```

经过 gcc 的预处理以后，以上面的 92 行为例，就变成了这样：

```
extern inline void outb(unsigned char value, unsigned short port) {
    __asm__ __volatile__ ("outb %b0, %w1" : : "a" (value), "Nd" (port)); }

extern inline void outb_p(unsigned char value, unsigned short port) {
    __asm__ __volatile__ ("outb %b0, %w1"
        "jmp 1f"
        "1: jmp 1f"
        "1:"
        : : "a" (value), "Nd" (port)); }
```

这里生成了两个函数，一个是 outb()，还有一个是 outb_p()。区别在于 outb_p()中在输出指令以后有意通过两条 jmp 指令引入了一些延迟。有些外设寄存器的速度比较慢，在写入以后需要一些恢复时间。如果 CPU 的速度太快，就有可能在对同一寄存器的连续两次写操作之间隔得太紧而使寄存器来不及恢复，在这种情况下就应该调用 outb_p()而不是 outb()，使寄存器有时间恢复。此外，汇编指令中的%b0 和%w1 表示%0 的宽度为 8 位，而%1 的宽度为 16 位。还可以看出，%b0 中的 b 是从 92 行传下来的，而%w1 中的 w 则是固定的（见 59 和 60 行），因为外设寄存器的地址是 16 位的。此外，变量 value 与寄存器%eax 结合，port 与%edx 结合。

同样，根据 93 和 94 行会分别生成出 outw()、outw_p()、outl()以及 outl_p()。

再看用于从寄存器输入的 inb()、inw()、inl()等函数的生成。

==================== include/asm-i386/io.h 82 90 ====================
```
82  #define RETURN_TYPE unsigned char
83  __IN(b,"")
84  #undef RETURN_TYPE
85  #define RETURN_TYPE unsigned short
86  __IN(w,"")
```

```
87   #undef RETURN_TYPE
88   #define RETURN_TYPE unsigned int
89   __IN(l,"")
90   #undef RETURN_TYPE
```

==================== include/asm-i386/io.h 62 70 ====================

```
62   #define __IN1(s) \
63   extern inline RETURN_TYPE in##s(unsigned short port) { RETURN_TYPE _v;
64
65   #define __IN2(s,s1,s2) \
66   __asm__ __volatile__ ("in" #s " %" s2 "1,%" s1 "0"
67
68   #define __IN(s,s1,i...) \
69   __IN1(s) __IN2(s,s1,"w") : "=a" (_v) : "Nd" (port) ,##i ); return _v; } \
70   __IN1(s##_p) __IN2(s,s1,"w") __FULL_SLOW_DOWN_IO : "=a" (_v) : "Nd" (port) ,##i ); return _v; } \
```

读者应该能推导出，根据 83 行的生成结果为：

```
unsigned char inb(unsigned short port) {
   unsigned char _v;
   __asm__ __volatile__ ("inb %w1, %0" : "=a" (v) : "Nd" (port),); return _v; }

unsigned char inb_p(unsigned short port) {
   unsigned char _v;
   __asm__ __volatile__ ("inb %w1, %0"
      "jmp 1f"
      "1: jmp 1f"
      "1:"
      : "a" (_v) : "Nd" (port),);
   return _v; }
```

这些底层函数在本章要阅读的代码中经常要碰到。

最后还要说明，由于设备的多样性，设备驱动是一个需要整本专著的大课题，甚至一本专著还不够。例如，在 drivers 目录下的子目录 net、arm、isdn 等都是计算机网络论述方面的内容，其中的每一部分都是需要有专著加以介绍，而不是三言两语讲得清的。与其语焉不详地讲上几句，倒不如留待将来，或等有关专著的出现，所以我们在本书中干脆就不触及这些话题。此外，有些设备本身的原理和机制就很复杂（如显示设备），需要有专著加以介绍。我们既无足够的篇幅，也缺乏有关的专门知识来深入阐述这些设备的原理、机制和操作，而只能专注于它们的驱动程序。所以，我们在本书中只能集中在除网络设备（那是一个过于广阔的天地）以外的一些典型设备的驱动程序上。但是，我们相信读者在理解了这些内容以后就能举一反三，将基本的原理与实现方法推广到其它设备。另外，在内核的代码中，对同一种设备常常因为针对具体产品而开发了大同小异的驱动程序，例如对鼠标器就有好几种驱动程序。如果我们选择了某个具体产品的驱动程序作为实例，那只不过是它的代码适合用作实例，或者我们碰巧选择了这些代码作为实例，而并不表示我们倾向于或偏爱该项具体的产品，更不表示我们倾向于生产该产品的厂商。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

8.2 系统调用 mknod()

“设备文件”是文件系统中代表设备的特殊文件。与普通的文件相比，设备文件在磁盘（或宿主文件系统所在的其它设备）上只占一个索引节点，而没有任何用于存放数据的记录块与之相联系。当然，这是因为设备文件的目的并不在于存储和读取数据，而只在于为应用程序提供一条通向具体设备的访问途径，使应用程序可以跟具体设备建立起连接。可想而知，既然没有用于数据的记录块，则索引节点中的记录块映象表（对于 Ext2 文件系统来说是 ext2_inode 结构中的数组 i_block[]）就没有什么用处了。所以，在这种情况下就用这个数组中的第一个元素，即 i_block[0]，记载目标设备的设备号。对于 Ext2 普通文件，这个数组中的元素在文件创建之初都是空的，只是在写文件时才随着记录块的分配而写入具体的内容（见“文件的读与写”）；而对于设备文件，则必须在创建时就将其所代表的设备号写入这个数组的第一个元素中。在“文件系统”一节中我们已经看到，普通文件（以及某些特殊文件）可以通过系统调用 open()创建，只要在调用参数中将 O_CREAT 标志位设成 1，就可以让 open()在目标文件不存在时先创建这个文件。此外，也可以通过系统调用 creat()创建文件，事实上 sys_creat()就是通过 sys_open()实现的。可是，这两个系统调用都不能用来创建设备文件，因为设备文件的创建需要有一个参数来传递设备号，而系统调用 open()和 creat()的界面都不包括这个参数。实际上，从 Unix 的早期就为设备文件的创建另外设置了一个系统调用 mknod()，并且一直沿用至今。

系统调用 mknod()是通用的，可以用来创建任何类型的文件（除目录外），包括普通文件、特殊文件以及设备文件。不过，由于其它类型的文件大都有专用的系统调用，如普通文件可以用 open()或 creat()创建，FIFO 文件可以用 pipe()创建，所以 mknod()主要用于设备文件的创建。此外，不像 open()和 creat()那样集创建与打开于一身，mknod()只是纯粹的创建。

在内核中，mknod()是由 sys_mknod()实现的，其代码在 fs/namei.c 中：

==================== fs/namei.c 1205 1246 ====================

```
asmlinkage long sys_mknod(const char * filename, int mode, dev_t dev)
{
	int error = 0;
	char * tmp;
	struct dentry * dentry;
	struct nameidata nd;

	if (S_ISDIR(mode))
		return -EPERM;
	tmp = getname(filename);
	if (IS_ERR(tmp))
		return PTR_ERR(tmp);

	if (path_init(tmp, LOOKUP_PARENT, &nd))
		error = path_walk(tmp, &nd);
	if (error)
		goto out;
	dentry = lookup_create(&nd, 0);
	error = PTR_ERR(dentry);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (!IS_ERR(dentry)) {
        switch (mode & S_IFMT) {
        case 0: case S_IFREG:
            error = vfs_create(nd.dentry->d_inode,dentry,mode);
            break;
        case S_IFCHR: case S_IFBLK: case S_IFIFO: case S_IFSOCK:
            error = vfs_mknod(nd.dentry->d_inode,dentry,mode,dev);
            break;
        case S_IFDIR:
            error = -EPERM;
            break;
        default:
            error = -EINVAL;
        }
        dput(dentry);
    }
    up(&nd.dentry->d_inode->i_sem);
    path_release(&nd);
out:
    putname(tmp);

    return error;
}
```

参数 dev 即为设备号，其类型为 dev_t，实际上是一个 16 位的无符号短整数，有关的定义见 include/asm-i386/posix_types.h 以及 include/linux/types.h：

===================== include/asm-i386/posix_types.h 10 10 =====================

```
typedef unsigned short __kernel_dev_t;
```

===================== include/linux/types.h 14 14 =====================

```
typedef __kernel_dev_t      dev_t;
```

这个 16 位的无符号整数实际上分成两部分，其高 8 位为表示设备类别的“主设备号”，而低 8 位则为同一类别内的序号。30 年前，256 种不同的块设备，再加 256 种不同的字符设备，每种设备又可以有多至 256 台，似乎已是不可思议的了。但是，现在再来看，8 位的主设备号和次设备号的覆盖面和容量确实显得小了。可是，在用户界面上，也就是在系统调用界面上，一旦把设备号定义为 16 位无符号整数后就不能改变了，除非能找到一种与原来兼容的方案，否则就要引起兼容性问题。在系统调用界面已经定义的前提下，解决的办法无非就是设法既扩大容量又使新老界面兼容（显然很难），或者增添新的系统调用（现在没有）或新的机制（如前述的树状设备文件子系统），或者甚至三管齐下。但是，对于内核中的内部使用却不受用户界面的限制。所以，作为扩大设备号容量的准备，又在 include/linux/kdev_t.h 中定义了另一个数据类型 kdev_t：

===================== include/linux/kdev_t.h 67 67 =====================

```
typedef unsigned short kdev_t;
```

这个数据类型目前仍是 16 位的，但是其意图显然是在条件成熟时改成 32 位。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在 sys_mknod()的代码中，只有 vfs_mknod()是读者尚未看到过的。文件系统的 vfs 层上同类的函数有三个，即 vfs_create()、vfs_mkdir()以及 vfs_mknod()。其中 vfs_create()用于普通文件，vfs_mkdir()用于目录节点，而 vfs_mknod()则用于特殊文件，包括 FIFO、插口以及字符设备和块设备文件的创建。至于/proc 目录下的特殊文件，则一般是由内核生成，而不是由用户创建的，这一点读者在“文件系统”一章中已经看到过了。

函数 vfs_mknod()的代码也在 fs/namei.c 中：

==================== fs/namei.c 1176 1203 ====================
[sys_mknod()>vfs_mknod()]

```
1176 int vfs_mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t dev)
1177 {
1178      int error = -EPERM;
1179
1180      mode &= ~current->fs->umask;
1181
1182      down(&dir->i_zombie);
1183      if ((S_ISCHR(mode) || S_ISBLK(mode)) && !capable(CAP_MKNOD))
1184           goto exit_lock;
1185
1186      error = may_create(dir, dentry);
1187      if (error)
1188           goto exit_lock;
1189
1190      error = -EPERM;
1191      if (!dir->i_op || !dir->i_op->mknod)
1192           goto exit_lock;
1193
1194      DQUOT_INIT(dir);
1195      lock_kernel();
1196      error = dir->i_op->mknod(dir, dentry, mode, dev);
1197      unlock_kernel();
1198 exit_lock:
1199      up(&dir->i_zombie);
1200      if (!error)
1201           inode_dir_notify(dir, DN_CREATE);
1202      return error;
1203 }
```

参数 dir 是一个 inode 结构指针，指向待创建设备文件的父节点，即其所在目录节点的 inode 结构，这是在 sys_mknod()中通过 path_walk()找到的。第二个参数 dentry，则指向代表着（如果已经存在）或将要代表待创建设备文件节点的目录项 dentry 数据结构。这个数据结构是在 sys_mknod()中通过 lookup_create()在内核中的 dentry 结构杂凑表中找到或者新创建的。

进入 vfs_mknod()以后，首先是对权限的检验。以前讲过，每个进程的 fs_struct 结构中有个访问权限屏蔽位图 umask，每当一个进程要求创建文件时，就用这个位图将用户通过参数 mode 表达的“非分之想”都过滤掉（1180 行）。对权限的检验分两步进行。第一步是 capable(CAP_MKNOD)，即检验当前进程是否允许创建设备节点，此项检验仅用于待创建节点为设备节点时（而不包括 FIFO 和插口节

PDF-XCHANGE Click to buy NOW! www.docu-track.com

点）。第二步，即 may_create()，则适用于所有节点，其代码读者已在“文件的打开与关闭”一节中见到过，此处就不重复了。这个函数中还包含了对目标节点是否业已存在的检验。显然，如果已经存在就不允许重复创建了。此外，由于我们在这里并不关心磁盘容量配额的问题，所以跳过 DQUOT_INIT()。

创建设备节点的具体操作因文件系统而异，由具体文件系统通过其 inode_operations 结构中的函数指针 mknod 提供。对于 Ext2，这个函数是 ext2_mknod()，其代码在 fs/ext2/namei.c 中：

==================== fs/ext2/namei.c 386 409 ====================

[sys_mknod()>vfs_mknod()>ext2_mknod()]

```
static int ext2_mknod (struct inode * dir, struct dentry *dentry, int mode, int rdev)
{
    struct inode * inode = ext2_new_inode (dir, mode);
    int err = PTR_ERR(inode);

    if (IS_ERR(inode))
        return err;

    inode->i_uid = current->fsuid;
    init_special_inode(inode, mode, rdev);
    err = ext2_add_entry (dir, dentry->d_name.name, dentry->d_name.len,
                 inode);
    if (err)
        goto out_no_entry;
    mark_inode_dirty(inode);
    d_instantiate(dentry, inode);
    return 0;

out_no_entry:
    inode->i_nlink--;
    mark_inode_dirty(inode);
    iput(inode);
    return err;
}
```

首先通过 ext2_new_inode()分配一个 inode 数据结构，这个函数的代码读者也已经看到过了。然后，就是设置这个数据结构，包括将设备号写入到该数据结构中，这是通过 init_special_inode()完成的。实际上，设备节点（以及其它特殊文件节点）的特殊之处也正在于此。这个函数的代码在 fs/devices.c 中：

==================== fs/devices.c 200 216 ====================

[sys_mknod()>vfs_mknod()>ext2_mknod()>init_special_inode()]

```
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = to_kdev_t(rdev);
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = to_kdev_t(rdev);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            inode->i_bdev = bdget(rdev);
        } else if (S_ISFIFO(mode))
            inode->i_fop = &def_fifo_fops;
        else if (S_ISSOCK(mode))
            inode->i_fop = &bad_sock_fops;
        else
            printk(KERN_DEBUG "init_special_inode: bogus imode (%o)\n", mode);
    }
```

这里主要是对两个结构成分的设置，一个是 file_operations 结构指针 i_fop，使它指向其体的数据结构 def_chr_fops 或 def_blk_fops；另一个是用来保存设备号的 i_rdev。注意在这里参数 rdev 的类型为 int（而不是 dev_t），而 inode 结构中 i_rdev 的类型就是上述的 kdev_t。目前，kdev_t 仍是 16 位的，从用户进程传递下来的设备号也是 16 位的，如果把它看成 32 位整数的话，则其高 16 位总是 0。但是将来在系统调用 mknod()的界面上可以把设备号的类型从 dev_t 改成 int，而根据其高 16 位是否为 0 确定所用的是 16 位设备号还是 32 位设备号，从而保持与原有界面兼容。同时，对 kdev_t 的定义也可以从 16 位改成 32 位。为了这个目的，内核代码中已经提供了一对 inline 函数 to_kdev_t()和 kdev_t_to_nr()，用来进行二者间的转换。但是目前这种转换只是从 16 位到 16 位，实际上不起什么作用，看一下 include/linux/kdev_t.h 中的函数 to_kdev_t()的代码就可以明白这一点：

==================== fs/devices.c 200 216 ====================

```
static inline kdev_t to_kdev_t(int dev)
{
    int major, minor;
#if 0
    major = (dev >> 16);
    if (!major) {
        major = (dev >> 8);
        minor = (dev & 0xff);
    } else
        minor = (dev & 0xffff);
#else
    major = (dev >> 8);
    minor = (dev & 0xff);
#endif
    return MKDEV(major, minor);
}
```

这里条件编译“#if 0”下面这一块就是为将来准备的，读一下这部分代码就可以明白 32 位的设备号怎样与 16 位的设备号保持兼容。

还要提一下，inode 结构中有两个类型为 kdev_t 的字段。一个是 i_rdev，用于该索引节点所代表设备（例如/dev/tty0）的设备号；另一个是 i_dev，用于索引节点所在设备（例如/dev/hda1，假定目录/dev 在这个设备上）的设备号。

对于块设备，inode 结构中还有个指针 i_bdev，它指向一个 block_device 数据结构，那就是具体块设备的控制结构，定义于 include/linux/fs.h：

==================== include/linux/fs.h 377 385 ====================

```
struct block_device {
	struct list_head	bd_hash;
	atomic_t		bd_count;
/*	struct address_space	bd_data; */
	dev_t			bd_dev;  /* not a kdev_t - it's a search key */
	atomic_t		bd_openers;
	const struct block_device_operations *bd_op;
	struct semaphore	bd_sem;	/* open/close mutex */
};
```

里面有个指针 bd_op 指向一个 block_device_operations 结构，那就是这个设备的函数跳转表。内核中为块设备设置了一个杂凑表 bdev_hashtable[]，每当首次创建或打开某个块设备文件时就为其创建一个 block_device 结构，并且根据设备号的杂凑值将此结构通过队列头 bd_hash 挂入杂凑表的某个队列中，以便查找。

在 init_special_inode()中，通过函数 bdget()根据设备号找到或者创建给定设备的 block_device 数据结构，其代码在 fs/block_dev.c 中：

==================== fs/block_dev.c 429 454 ====================

[sys_mknod()>vfs_mknod()>ext2_mknod()>init_special_inode()>bdget()]

```
429 struct block_device *bdget(dev_t dev)
430 {
431 	struct list_head * head = bdev_hashtable + hash(dev);
432 	struct block_device *bdev, *new_bdev;
433 	spin_lock(&bdev_lock);
434 	bdev = bdfind(dev, head);
435 	spin_unlock(&bdev_lock);
436 	if (bdev)
437 		return bdev;
438 	new_bdev = alloc_bdev();
439 	if (!new_bdev)
440 		return NULL;
441 	atomic_set(&new_bdev->bd_count,1);
442 	new_bdev->bd_dev = dev;
443 	new_bdev->bd_op = NULL;
444 	spin_lock(&bdev_lock);
445 	bdev = bdfind(dev, head);
446 	if (!bdev) {
447 		list_add(&new_bdev->bd_hash, head);
448 		spin_unlock(&bdev_lock);
449 		return new_bdev;
450 	}
451 	spin_unlock(&bdev_lock);
452 	destroy_bdev(new_bdev);
453 	return bdev;
454 }
```

首先是试着在杂凑表 bdev_hashtable[]里的相应队列中通过 bdfind()寻找，如果找到就完成了（见 437 行）。要是找不到，那就要通过 alloc_bdev()分配一个数据结构。由于在 alloc_bdev()的过程中当前

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

进程有可能进入睡眠，所以在分配到了以后还要再调用 bdfind()寻找一遍，以防因别的进程抢先为同一设备分配了数据结构而造成冲突。在确认并未造成冲突以后，就将该数据结构挂入相应的杂凑队列中。值得注意的是，这里只是将数据结构中的指针 bd_op 初始化成 NULL，而将这个指针的落实留待第一次打开这个设备文件的时候。读者在“文件系统的安装与拆卸”一节中曾看到，在安装文件系统时通过 get_blkfops()根据设备的主设备号取得指向其 block_device_operations 结构的指针，就是因为在安装时隐含着打开块设备文件的操作。

回到 ext2_mknod()的代码中，ext2_add_entry()将新创建的节点加进所在目录在磁盘上的目录文件中，mark_inode_dirty()将新创建的 inode 结构设置成“脏”，而 d_instantiate()则使内存中的目录项 dentry 结构与 inode 结构间互相挂上钩，这些函数读者都已不是第一次看到了。由于新创建的 inode 结构设置成了“脏”，内核在“同步”内存中的 inode 结构与磁盘上的索引节点的时候，就会将这个 inode 结构的内容写到磁盘上分配给这个文件的索引节点，即 ext2_inode 数据结构中。由于 ext2_inode 结构中并不存在 i_rdev 这么个成分，而对于设备文件却又不需要使用 i_block[]数组，所以就挪用其 i_block[0]来保存设备号。要了解这一点，我们不妨看一下文件 fs/ext2/inode.c 中函数 ext2_update_inode()代码的一个片段：

==================== fs/ext2/inode.c 1211 1214 ====================

```
	if (S_ISCHR(inode->i_mode) || S_ISBLK(inode->i_mode))
		raw_inode->i_block[0] = cpu_to_le32(kdev_t_to_nr(inode->i_rdev));
	else for (block = 0; block < EXT2_N_BLOCKS; block++)
		raw_inode->i_block[block] = inode->u.ext2_i.i_data[block];
```

这里的 raw_inode 是一个 ext2_inode 结构指针，而 inode 是一个 inode 结构指针。这二者一个是存储在磁盘上的索引节点，另一个是索引节点在内存中的表现，但是二者在内容上并不完全相同，inode 结构中含有许多动态的信息。如果 inode 结构代表着一个字符设备或块设备，则相应 ext2_inode 结构中的 i_block[0]用于存放设备号，其内容来自 inode 结构中的 i_rdev（须转换成“Little Ending”）。要不然，则 i_block[]数组中各个元素的内容来自 inode 结构中 ext2_inode_info 部分的 i_data[]数组的相应元素，那是一些直接或间接地指向各个数据记录块的指针。

反过来，当通过 ext2_read_inode()从磁盘上读入索引节点，并为之在内存中创建相应的 inode 结构时，则先将 i_block[]数组全部复制到 i_data[]数组中，然后如果是设备文件就调用 init_special_inode()将 i_block[0]的内容（从“Little Ending”转换回 CPU 所采用的制式）填入 inode 结构中的 i_rdev。以下是 ext2_read_inode()中有关的片断（见 fs/ext2/inode.c）：

==================== fs/ext2/inode.c 1052 1062 ====================

```
	/*
	 * NOTE! The in-memory inode i_data array is in little-endian order
	 * even on big-endian machines: we do NOT byteswap the block numbers!
	 */
	for (block = 0; block < EXT2_N_BLOCKS; block++)
		inode->u.ext2_i.i_data[block] = raw_inode->i_block[block];

	if (inode->i_ino == EXT2_ACL_IDX_INO ||
	   inode->i_ino == EXT2_ACL_DATA_INO)
		/* Nothing to do */ ;
```

```
     else if (S_ISREG(inode->i_mode)) {
  . . . . . .
```

==================== fs/ext2/inode.c 1066 1066 ====================

```
     } else if (S_ISDIR(inode->i_mode)) {
  . . . . . .
```

==================== fs/ext2/inode.c 1069 1069 ====================

```
     } else if (S_ISLNK(inode->i_mode)) {
  . . . . . .
```

==================== fs/ext2/inode.c 1076 1078 ====================

```
     } else
          init_special_inode(inode, inode->i_mode,
                      le32_to_cpu(raw_inode->i_block[0]));
```

这就又回到了上面的 init_special_inode()。

对于设备文件，将 ext2_inode 结构中的 i_block[]数组复制到 inode 结构中的 i_data[]数组实际上是没有意义的。至于 inode 结构中的 block_device 结构指针 i_bdev，则本来就是动态的，所以根本就不保存在磁盘上的索引节点中。

8.3 可安装模块

早在第 1 章我们就曾提到，Linux 采用的是整体式的内核结构（monolithic kernel），这种结构的内核一般不能动态地增加新的功能。为此，Linux 提供了一种全新的机制，叫（可安装）“模块”（module）。利用这个机制，可以根据需要，在不必对内核重新编译连接的条件下，将可安装模块动态地插入运行中的内核，成为内核的一个有机组成部分；或者从内核移走已经安装的模块。正是这种机制，使得内核的内存映象保持最小，但却具有很大的灵活性和可扩充性。

可安装模块是可以在系统运行时动态地安装和拆卸的内核软件。严格说来，这种软件的作用并不限于设备驱动，例如有些文件系统就是以可安装模块的形式实现的。但是，另一方面，它主要用来实现设备驱动程序或者与设备驱动密切相关的部分（如文件系统），所以我们将它放在本章中介绍。

在应用程序界面上，内核通过 4 个系统调用支持可安装模块的动态安装和拆卸，它们是 create_module()、init_module()、query_module()以及 delete_module()。通常，用户一般不需要直接跟这些系统调用打交道，而可以用系统提供的工具/sbin/insmod（插入模块）和/sbin/rmmod（移走模块）来安装和拆卸可安装模块。当然，这两个工具最终还是要通过这些系统调用完成有关操作。大体上说，/sbin/insmod 所做的事情有这么一些：

- 打开待安装模块并将其读入到用户空间。所谓“模块”就是经过编译但未经连接的.o 文件。
- 模块中必定有一些在模块内部无法落实的符号（函数名或变量名），对这些符号的引用必须连接到内核中的相应符号，也就是必须把这些符号在内核映象中的地址填入模块中需要访问这些符号的指令中以及数据结构中。为此目的，需要通过系统调用 query_module()向内核询问这些符号在内核中的地址。如果内核允许“移出”这些符号的地址，就会返回有关的“符号表”。有些符号可能并不属于内核本身，而属于已经安装的其它模块。
- 取得了内核“移出”的符号表以后，就应该可以使模块中所有的符号引用都得到落实了。这

PDF-XCHANGE Click to buy NOW! www.docu-track.com

部分操作与编译后的连接相似。不过，常规的“连接”常常是双向的，而现在只是在模块中引用内核里的符号或者某些已经安装的模块中的符号，而内核却并不要求（也不能）反过来引用这个待安装模块中的符号。当然，内核最后一定会要访问模块中的某些变量或调用模块中的某些函数，否则模块中的函数就得不到执行，模块的存在也就失去了意义。从这个意义上说，模块与内核的连接只完成了一半，我们不妨称之为“完成了单向连接”的模块映象。

- 然后，通过系统调用 create_module()在内核中创建一个 module 数据结构，并且“预订”所需的系统（内核）空间。
- 最后，通过系统调用 init_module()把用户空间中完成了单向连接的模块映象装入内核空间，再调用模块中一个名为 init_module()的函数。注意，不要把可安装模块中的函数 init_module()与系统调用 init_module()搞混淆了，这完全是两码事。系统调用 init_module()在内核中的实现是 sys_init_module()，这是由内核提供的，整个内核中只有这么一个。而模块中的函数 init_module()，则是由可安装模块本身提供的，每个模块都有这样一个函数。通常，每个模块的 init_module()负责向内核“登记”本模块中的一些包含着函数指针的数据结构（例如 file_operations 结构）。完成了这种登记以后，模块与内核之间的连接（另一个方向上的连接）才真正完成了。

顾名思义，系统调用 delete_module()将模块的 module 结构释放，并且将模块映象所占的内核空间释放。此外，还有一件重要的事情就是调用模块中一个名为 cleanup_module()的函数。与 init_module()函数一样，每个可安装模块都有其自己的 cleanup_module()函数。通常，在这个函数中要向内核撤销对由本模块提供的数据结构（从而函数指针）的登记，使内核在模块拆卸以后不至于再企图访问这些数据结构（以及函数指针）。

下面，我们先看 4 个系统调用的代码，然后再通过一个实际的模块来看一下典型的 init_module()函数和 cleanup_module()函数。函数 sys_create_module()、sys_init_module()、sys_query_module()，还有 sys_delete_module()的代码都在 kernel/module.c 中。

先看 sys_query_module()。

==================== kernel/module.c 874 929 ====================

```
asmlinkage long
sys_query_module(const char *name_user, int which, char *buf, size_t bufsize,
          size_t *ret)
{
     struct module *mod;
     int err;

     lock_kernel();
     if (name_user == NULL)
          mod = &kernel_module;
     else {
          long namelen;
          char *name;

          if ((namelen = get_mod_name(name_user, &name)) < 0) {
               err = namelen;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            goto out;
        }
        err = -ENOENT;
        if (namelen == 0)
            mod = &kernel_module;
        else if ((mod = find_module(name)) == NULL) {
            put_mod_name(name);
            goto out;
        }
        put_mod_name(name);
    }

    switch (which)
    {
    case 0:
        err = 0;
        break;
    case QM_MODULES:
        err = qm_modules(buf, bufsize, ret);
        break;
    case QM_DEPS:
        err = qm_deps(mod, buf, bufsize, ret);
        break;
    case QM_REFS:
        err = qm_refs(mod, buf, bufsize, ret);
        break;
    case QM_SYMBOLS:
        err = qm_symbols(mod, buf, bufsize, ret);
        break;
    case QM_INFO:
        err = qm_info(mod, buf, bufsize, ret);
        break;
    default:
        err = -EINVAL;
        break;
    }
out:
    unlock_kernel();
    return err;
}
```

参数 name_user 为查询对象所在模块的模块名，如果这个指针为 0 即指内核本身。参数 which 表示查询的内容，如 QM_SYMBOLS 表示查询目标模块的符号表，QM_MODULES 表示查询内核中已安装模块的清单，而 QM_DEPS 则查询给定模块对其它模块的依赖（如果模块 A 引用模块 B 中的符号，则 A 依赖于 B）。查询的结果通过缓冲区 buf 返回。

每个已安装模块在内核中都有一个 module 数据结构（通过系统调用 create_module()创建），include/linux/module.h 中定义了 module 以及一些有关的数据结构：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/module.h 37 93 ====================

```
struct module_symbol
{
        unsigned long value;
        const char *name;
};

struct module_ref
{
        struct module *dep;     /* "parent" pointer */
        struct module *ref;     /* "child" pointer */
        struct module_ref *next_ref;
};

/* TBD */
struct module_persist;

struct module
{
        unsigned long size_of_struct;    /* == sizeof(module) */
        struct module *next;
        const char *name;
        unsigned long size;

        union
        {
                atomic_t usecount;
                long pad;
        } uc;                   /* Needs to keep its size - so says rth */

        unsigned long flags;        /* AUTOCLEAN et al */

        unsigned nsyms;
        unsigned ndeps;

        struct module_symbol *syms;
        struct module_ref *deps;
        struct module_ref *refs;
        int (*init)(void);
        void (*cleanup)(void);
        const struct exception_table_entry *ex_table_start;
        const struct exception_table_entry *ex_table_end;
#ifdef __alpha__
        unsigned long gp;
#endif
        /* Members past this point are extensions to the basic
           module support and are optional.  Use mod_member_present()
           to examine them.  */
        const struct module_persist *persist_start;
```

```
        const struct module_persist *persist_end;
        int (*can_unload)(void);
        int runsize;            /* In modutils, not currently used */
        const char *kallsyms_start; /* All symbols for kernel debugging */
        const char *kallsyms_end;
        const char *archdata_start; /* arch specific data for module */
        const char *archdata_end;
        const char *kernel_data;    /* Reserved for kernel internal use */
};
```

这里主要定义了三种数据结构，第一种数据结构是 module_symbol，每个 module_symbol 结构描述了一个符号，包括符号名及其所在的地址，或者说符号的值。第二种是 module_ref，用来描述模块间的依赖关系。最后就是 module 结构。内核中的 module 结构通过它们的指针 next 连成一个链。每个模块都有个模块名 name，size 则为模块的大小，函数指针 init 和 cleanup 分别指向模块的 init_module()和 cleanup_module()函数。指针 syms 指向一个 module_symbol 结构数组，而 nsyms 指明了数组的大小，此即为模块的符号表。对模块中的具体符号是否放在其符号表中也是可以选择的，对此我们还将详细介绍。指针 deps 指向一个 module_ref 结构数组，ndeps 则为该数组的大小。这个数组中的每一个元素都通过其指针 dep 指向一个 module 结构，这些就是给定模块所依赖的模块。要安装一个模块时，这个模块所依赖的所有模块都必须已经安装好。与依赖关系（dep）方向相反的是“被引用”关系 ref。就是说，如果模块 A 引用了模块 B 中的符号，则 A 依赖于 B，或者说 B 被 A 引用。注意这里的所谓“依赖”，即“ dep”，实际上就是“引用”，只不过是主动语态的“引用”，而“ ref”则是被动语态的“引用”，所以二者方向相反。

如果 A 依赖于 B，则模块 A 的 deps 数组中有一个元素指向模块 B。但是，光有这样的联系还不够方便，假如某一个应用程序要拆卸模块 B，那就要扫描所有已安装模块的 deps 数组才能知道是否还有模块依赖于 B（如果有，则 B 还有“用户”而不能拆除）。所以，在 module 结构中还有个指针 refs，它指向一个 module_ref 结构链，链中的每个结构都通过指针 ref 指向一个依赖于它的模块，并且就在那个模块的 deps 数组中。在我们这个例子中，模块 A 的 deps 数组中就应该有这么一个 module_ref 结构，其指针 dep 指向 B；指针 ref 则指向 A 自身，并且通过链接指针 next_ref 链入 B 的 refs 链中。这样，从模块 A 出发，通过其 deps 数组就能找到 A 所依赖的所有模块，包括 B 在内。反过来，从模块 B 出发，则通过其 refs 链就能找到所有引用了 B 的模块，包括 A 在内。这里，“依赖”关系是静态的，一个模块依赖于哪几个模块是固定的，模块一经实现，这一点就定下来了，所以适合于使用数组。而反过来，“被引用”关系则是动态的，一个模块被几个模块引用随时都可能变动，所以要采用动态的 module_ref 结构链（即队列）。

虽然内核并不是可安装模块，但是它也有符号表，也受到其它模块的引用，将其看成可安装模块可以简化程序设计，所以为内核也定义了一个 module 结构，从 kernel_module 开始，所有已安装模块的 module 结构都链接在一起成为一条链，内核中的全局量 module_list 就指向这条链。

此处建议读者先不忙往下看，先把前面介绍的三个数据结构以及相互之间的联系整理一下，画张图。这对理解模块机制源代码会有好处。

理解了这几个数据结构，sys_query_module()的代码就很简单了。如前所述，如果参数 name_user 为 NULL 就表示查询的对象为内核本身，其 module 结构即 kernel_module，这是在 kernel/module.c 中

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

定义的：

==================== kernel/module.c 41 52 ====================
```
static struct module kernel_module =
{
    size_of_struct:         sizeof(struct module),
    name:                   "",
    uc:             {ATOMIC_INIT(1)},
    flags:                  MOD_RUNNING,
    syms:                   __start___ksymtab,
    ex_table_start:         __start___ex_table,
    ex_table_end:       __stop___ex_table,
    kallsyms_start:         __start___kallsyms,
    kallsyms_end:       __stop___kallsyms,
};
```

凡是不出现在初始值定义中的字段（如 deps 和 refs 等等），其值均为 0 或 NULL。显然，内核没有 init_module()和 cleanup_module()这两个函数，因为相应的函数指针都是 NULL。同时，内核没有 deps 数组，开始时也没有 refs 链。可是，这个数据结构中的指针 syms 指向__start__ksymtab，这就是内核符号表的起始地址。符号表的大小 nsyms 为 0，但是在系统初始化时会在函数 init_modules()中将其设置成正确的数值。

如果参数 name_user 不是 NULL，那就是查询一个特定的模块了。所以在通过 get_mod_name()从用户空间复制模块名以后就通过 find_module()在队列 module_list 中寻找相应的 module 结构。这个函数的代码也在 kernel/module.c 中：

==================== kernel/module.c 993 1010 ====================
[sys_query_module()>find_module()]
```
/*
 * Look for a module by name, ignoring modules marked for deletion.
 */

struct module *
find_module(const char *name)
{
    struct module *mod;

    for (mod = module_list; mod ; mod = mod->next) {
        if (mod->flags & MOD_DELETED)
            continue;
        if (!strcmp(mod->name, name))
            break;
    }

    return mod;
}
```

至于对具体查询的服务，即 qm_modules()、qm_symbols()、qm_deps()等等，则很简单了。我们看

一下 kernel/module.c 中函数 qm_deps()的代码：

==================== kernel/module.c 701 742 ====================
[sys_query_module()>qm_deps()]

```
static int
qm_deps(struct module *mod, char *buf, size_t bufsize, size_t *ret)
{
        size_t i, space, len;

        if (mod == &kernel_module)
                return -EINVAL;
        if (!MOD_CAN_QUERY(mod))
                if (put_user(0, ret))
                        return -EFAULT;
                else
                        return 0;

        space = 0;
        for (i = 0; i < mod->ndeps; ++i) {
                const char *dep_name = mod->deps[i].dep->name;

                len = strlen(dep_name)+1;
                if (len > bufsize)
                        goto calc_space_needed;
                if (copy_to_user(buf, dep_name, len))
                        return -EFAULT;
                buf += len;
                bufsize -= len;
                space += len;
        }

        if (put_user(i, ret))
                return -EFAULT;
        else
                return 0;

calc_space_needed:
        space += len;
        while (++i < mod->ndeps)
                space += strlen(mod->deps[i].dep->name)+1;

        if (put_user(space, ret))
                return -EFAULT;
        else
                return -ENOSPC;
}
```

系统调用返回 0 表示成功，此时参数 ret 所指的整数含有所依赖模块的个数，而 buf 中则为这些模块的模块名。否则表示执行失败，出错代码为-ENOSPC 表示 buf 的空间太小，此时 ret 所指的整数含

有所需的空间大小。这些代码就不需要解释了，其它几个也是一样，有兴趣的读者可以自己阅读。

再看系统调用 create_module()的实现 sys_create_module()，也在 kernel/module.c 中：

==================== kernel/module.c 276 325 ====================

```
/*
 * Allocate space for a module.
 */

asmlinkage unsigned long
sys_create_module(const char *name_user, size_t size)
{
     char *name;
     long namelen, error;
     struct module *mod;

     if (!capable(CAP_SYS_MODULE))
          return -EPERM;
     lock_kernel();
     if ((namelen = get_mod_name(name_user, &name)) < 0) {
          error = namelen;
          goto err0;
     }
     if (size < sizeof(struct module)+namelen) {
          error = -EINVAL;
          goto err1;
     }
     if (find_module(name) != NULL) {
          error = -EEXIST;
          goto err1;
     }
     if ((mod = (struct module *)module_map(size)) == NULL) {
          error = -ENOMEM;
          goto err1;
     }

     memset(mod, 0, sizeof(*mod));
     mod->size_of_struct = sizeof(*mod);
     mod->next = module_list;
     mod->name = (char *)(mod + 1);
     mod->size = size;
     memcpy((char*)(mod+1), name, namelen+1);

     put_mod_name(name);

     module_list = mod; /* link it in */

     error = (long) mod;
     goto err0;
err1:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
321       put_mod_name(name);
322  err0:
323       unlock_kernel();
324       return error;
325  }
```

只有特权用户才允许在内核中创建模块，capable(CAP_SYS_MODULE)就是对当前进程是否有这种特权的检查。参数 size 表示模块的大小，包括模块的映象本身以及一个 module 结构的大小，再加上模块名的长度。显然，size 至少也得等于后两者的和，否则就肯定错了。模块是以模块名来惟一地加以标识的，所以先要通过 find_module()检查，看是否已经有同名的模块存在。通过了所有这些检验以后，就调用 module_map()分配空间。对于 i386 结构的 CPU，module_map()在 include/asm-i386/module.h 中定义为 vmalloc()：

==================== include/asm-i386/module.h 7 7 ====================

```
7    #define module_map(x)        vmalloc(x)
```

分配得的空间的开头用作模块的 module 数据结构（见 307 行），然后是模块名字符串（见 312 行）。注意这里(mod+1)表示从 mod 所指的地址加上一个 module 结构大小的地方。最后剩下的用于模块的映象。显然，新建立的 module 结构基本上是空的，其内容有待于从用户空间传递过来。

对模块的 module 结构加以初始化，并将其链入 module_list 前端（见 309 行和 316 行）。此后，创建模块的操作就完成了，系统调用返回内核中为这个模块所分配的空间地址。下一步是系统调用 init_module()。

如前所述，在系统调用 init_module()之前要由应用程序在用户空间完成模块与内核符号间的连接，这个过程与“连接”，即 ld 的过程相似。由于这是在用户空间完成的，不属于内核的活动，所以我们不深入到这个过程中去了，有兴趣的读者可以从 GNU 网站下载 insmod 的源程序自己阅读。

相比之下，sys_init_module()的代码（见 kernel/module.c）就比较大了，我们分段阅读。

==================== kernel/module.c 327 390 ====================

[sys_init_module()]

```
327  /*
328   * Initialize a module.
329   */
330
331  asmlinkage long
332  sys_init_module(const char *name_user, struct module *mod_user)
333  {
334       struct module mod_tmp, *mod;
335       char *name, *n_name, *name_tmp = NULL;
336       long namelen, n_namelen, i, error;
337       unsigned long mod_user_size;
338       struct module_ref *dep;
339
340       if (!capable(CAP_SYS_MODULE))
341            return -EPERM;
342       lock_kernel();
343       if ((namelen = get_mod_name(name_user, &name)) < 0) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
		error = namelen;
		goto err0;
	}
	if ((mod = find_module(name)) == NULL) {
		error = -ENOENT;
		goto err1;
	}

	/* Check module header size.  We allow a bit of slop over the
	   size we are familiar with to cope with a version of insmod
	   for a newer kernel.  But don't over do it. */
	if ((error = get_user(mod_user_size, &mod_user->size_of_struct)) != 0)
		goto err1;
	if (mod_user_size < (unsigned long)&((struct module *)0L)->persist_start
	    || mod_user_size > sizeof(struct module) + 16*sizeof(void*)) {
		printk(KERN_ERR "init_module: Invalid module header size.\n"
		       KERN_ERR "A new version of the modutils is likely "
				"needed.\n");
		error = -EINVAL;
		goto err1;
	}

	/* Hold the current contents while we play with the user's idea
	   of righteousness.  */
	mod_tmp = *mod;
	name_tmp = kmalloc(strlen(mod->name) + 1, GFP_KERNEL);  /* Where's kstrdup()? */
	if (name_tmp == NULL) {
		error = -ENOMEM;
		goto err1;
	}
	strcpy(name_tmp, mod->name);

	error = copy_from_user(mod, mod_user, mod_user_size);
	if (error) {
		error = -EFAULT;
		goto err2;
	}

	/* Sanity check the size of the module.  */
	error = -EINVAL;

	if (mod->size > mod_tmp.size) {
		printk(KERN_ERR "init_module: Size of initialized module "
				"exceeds size of created module.\n");
		goto err2;
	}

```

首先还是对权限的检验。接着就是从用户空间复制模块名，并通过 find_module()从 module_list 找

PDF-XCHANGE Click to buy NOW! www.docu-track.com

到目标模块的 module 数据结构。在系统调用 init_module()之前，应用程序要在用户空间为目标模块准备一个 module 数据结构，并且将指向这个结构的指针作为参数传给内核，这就是这里的指针 mod_user。此时内核中虽然已经由 sys_create_module()创建了目标模块的 module 数据结构，但是这个数据结构基本上还是空的，其内容只能来自用户空间。现在，就要把用户空间 module 结构的内容复制到内核中的对应 module 结构中。可是，由于版本等等的原因，用户空间的 module 结构有可能与内核中的不完全一样。为了防止因用户空间的 module 结构与内核中的 module 结构大小不符而造成麻烦，先把用户空间 module 结构中的 size_of_struct 字段复制过来加以检查。在 module 数据结构的定义中，从 persist_start 开始的三个指针是内核对这个数据结构的扩充，用户空间的 module 结构有可能并不包括这些字段（读者不妨试一下“ man init_module”看看），所以检查其大小时的条件之一是不小于结构中 persist_start 以前的那一部分。同时，这个大小也不能超过内核中 module 结构的大小加上 16 个指针，即 64 个字节的位置。通过了对结构大小的检查以后，先把内核中的 module 结构暂时保存在堆栈中作为后备（见 368 行），然后就从用户空间复制其 module 结构。复制时是以内存中的 module 结构大小为准的，以免“冲坏”内核中的内存空间。复制过来以后，还要核对“新版”数据结构中指明的模块大小与原来“预约 ”的模块大小是否一致（见 385 行）。

通过了对结构大小的检验以后，下一步就是检查结构中内容的合理性。继续往下看：

==================== kernel/module.c 391 483 ====================

[sys_init_module()]

```
391      /* Make sure all interesting pointers are sane.  */
392
393      if (!mod_bound(mod->name, namelen, mod)) {
394           printk(KERN_ERR "init_module: mod->name out of bounds.\n");
395           goto err2;
396      }
397      if (mod->nsyms && !mod_bound(mod->syms, mod->nsyms, mod)) {
398           printk(KERN_ERR "init_module: mod->syms out of bounds.\n");
399           goto err2;
400      }
401      if (mod->ndeps && !mod_bound(mod->deps, mod->ndeps, mod)) {
402           printk(KERN_ERR "init_module: mod->deps out of bounds.\n");
403           goto err2;
404      }
405      if (mod->init && !mod_bound(mod->init, 0, mod)) {
406           printk(KERN_ERR "init_module: mod->init out of bounds.\n");
407           goto err2;
408      }
409      if (mod->cleanup && !mod_bound(mod->cleanup, 0, mod)) {
410           printk(KERN_ERR "init_module: mod->cleanup out of bounds.\n");
411           goto err2;
412      }
413      if (mod->ex_table_start > mod->ex_table_end
414          || (mod->ex_table_start &&
415           !((unsigned long)mod->ex_table_start >= ((unsigned long)mod + mod->size_of_struct)
416             && ((unsigned long)mod->ex_table_end
417                 < (unsigned long)mod + mod->size))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            || (((unsigned long)mod->ex_table_start
              - (unsigned long)mod->ex_table_end)
             % sizeof(struct exception_table_entry))) {
             printk(KERN_ERR "init_module: mod->ex_table_* invalid.\n");
             goto err2;
        }
        if (mod->flags & ~MOD_AUTOCLEAN) {
             printk(KERN_ERR "init_module: mod->flags invalid.\n");
             goto err2;
        }
#ifdef __alpha__
        if (!mod_bound(mod->gp - 0x8000, 0, mod)) {
             printk(KERN_ERR "init_module: mod->gp out of bounds.\n");
             goto err2;
        }
#endif
        if (mod_member_present(mod, can_unload)
            && mod->can_unload && !mod_bound(mod->can_unload, 0, mod)) {
             printk(KERN_ERR "init_module: mod->can_unload out of bounds.\n");
             goto err2;
        }
        if (mod_member_present(mod, kallsyms_end)) {
            if (mod->kallsyms_end &&
             (!mod_bound(mod->kallsyms_start, 0, mod) ||
              !mod_bound(mod->kallsyms_end, 0, mod))) {
             printk(KERN_ERR "init_module: mod->kallsyms out of bounds.\n");
             goto err2;
            }
            if (mod->kallsyms_start > mod->kallsyms_end) {
             printk(KERN_ERR "init_module: mod->kallsyms invalid.\n");
             goto err2;
            }
        }
        if (mod_member_present(mod, archdata_end)) {
            if (mod->archdata_end &&
             (!mod_bound(mod->archdata_start, 0, mod) ||
              !mod_bound(mod->archdata_end, 0, mod))) {
             printk(KERN_ERR "init_module: mod->archdata out of bounds.\n");
             goto err2;
            }
            if (mod->archdata_start > mod->archdata_end) {
             printk(KERN_ERR "init_module: mod->archdata invalid.\n");
             goto err2;
            }
        }
        if (mod_member_present(mod, kernel_data) && mod->kernel_data) {
            printk(KERN_ERR "init_module: mod->kernel_data must be zero.\n");
            goto err2;
        }
```

```
467
468     /* Check that the user isn't doing something silly with the name.  */
469
470     if ((n_namelen = get_mod_name(mod->name - (unsigned long)mod
471                        + (unsigned long)mod_user,
472                        &n_name)) < 0) {
473         printk(KERN_ERR "init_module: get_mod_name failure.\n");
474         error = n_namelen;
475         goto err2;
476     }
477     if (namelen != n_namelen || strcmp(n_name, mod_tmp.name) != 0) {
478         printk(KERN_ERR "init_module: changed module name to "
479                 "`%s' from `%s'\n",
480               n_name, mod_tmp.name);
481         goto err3;
482     }
483
```

这里的宏操作 mod_bound()用来检查由用户提供的指针所指的对象是否落在模块的边界内，定义于 include/linux/module.h。

==================== include/linux/module.h 133 135 ====================

```
/* Check if an address p with number of entries n is within the body of module m */
#define mod_bound(p, n, m) ((unsigned long)(p) >= ((unsigned long)(m) + ((m)->size_of_struct)) && \
            (unsigned long)((p)+(n)) <= (unsigned long)(m) + (m)->size)
```

以第 393 行对模块名的检验为例，要检查的是 module 结构中的指针 name 指向为该模块分配的缓冲区内部，但是又不落在 module 结构的内部，同时，长度为 namelen 的字符串又不越出模块缓冲区的边界。简言之，这个字符串从必须在模块映象的范围内。类似地，如果模块的符号表以及 deps 数组非空，则也必须落在为模块分配的缓冲区中、module 结构外。不光是对指向数据结构的指针，对函数指针也是一样，init_module()和 cleanup_module()两个函数必须在模块映象的范围中。

在模块映象中也可以包含对异常的处理。发生于一些特殊地址上的异常，可以通过一种描述结构 exception_table_entry 规定对异常的反应和处理，这些描述结构在可执行映象连接时都被集中在一个数组中（内核的 exception_table_entry 结构数组为__start___ex_table[]）。当异常发生时，内核的异常响应程序会先搜索这个数组，看看是否对所发生的异常（根据异常发生时的地址）规定了特殊的处理。我们在第 3 章中曾讲到过有关的情景，读者可以回过去看一下。当内核支持可安装模块时，其异常响应程序会扫描 module_list，对于已安装的每个模块都搜索其异常处理描述表（如前所述，内核本身也被看作是一个模块）。理所当然，每个模块的异常处理描述表也必须在模块映象的范围内。虽然对异常处理描述表的检验没有使用宏操作 bound，但是其精神是一致的。之所以在这里不使用 bound，是因为异常处理描述表是以起点和终点而不是起点加长度来界定其范围的。

内核中 module 结构内的最后一个字段是函数指针 can_unload，但是这属于 module 结构的扩充部分，用户空间的 module 结构可能包括也可能不包括这个字段。如果包括了这个字段，那就也要保证这个指针指向模块的映象内部。那么，怎样才能知道用户空间的 module 结构是否包括这个字段呢？方法是检查这个结构的大小，为此目的定义了一个宏操作 mod_member_present，它是在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

include/linux/module.h 中定义的：

==================== include/linux/module.h 125 131 ====================

```
/* When struct module is extended, we must test whether the new member
   is present in the header received from insmod before we can use it.
   This function returns true if the member is present.  */

#define mod_member_present(mod,member)                              \
      ((unsigned long)(&((struct module *)0L)->member + 1)         \
       <= (mod)->size_of_struct)
```

最后，对于模块名还要作一番检验。虽然在前面已经根据参数 name_user 从用户空间复制了作为系统调用参数的模块名，但是这个模块名是否与用户空间 module 结构中所指示的模块名一致呢？显然，不能排除不一致的可能性，所以现在要根据 module 结构的内容把模块映象中的模块名也复制过来，再与原先使用的模块名比较（见 470 至 477 行）。

经过了所有这些检验以后，可以从用户空间把模块的映象复制过来了。我们往下看：

==================== kernel/module.c 484 540 ====================

[sys_init_module()]

```
	/* Ok, that's about all the sanity we can stomach; copy the rest.  */

	if (copy_from_user((char *)mod+mod_user_size,
			   (char *)mod_user+mod_user_size,
			   mod->size-mod_user_size)) {
		error = -EFAULT;
		goto err3;
	}

	if (module_arch_init(mod))
		goto err3;

	/* On some machines it is necessary to do something here
	   to make the I and D caches consistent.  */
	flush_icache_range((unsigned long)mod, (unsigned long)mod + mod->size);

	mod->next = mod_tmp.next;
	mod->refs = NULL;

	/* Sanity check the module's dependents */
	for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
		struct module *o, *d = dep->dep;

		/* Make sure the indicated dependencies are really modules.  */
		if (d == mod) {
			printk(KERN_ERR "init_module: self-referential "
			       "dependency in mod->deps.\n");
			goto err3;
		}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
513
514             /* Scan the current modules for this dependency */
515             for (o = module_list; o != &kernel_module && o != d; o = o->next)
516                 ;
517
518             if (o != d) {
519                 printk(KERN_ERR "init_module: found dependency that is "
520                     "(no longer?) a module.\n");
521                 goto err3;
522             }
523         }
524
525         /* Update module references.  */
526         for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
527             struct module *d = dep->dep;
528
529             dep->ref = mod;
530             dep->next_ref = d->refs;
531             d->refs = dep;
532             /* Being referenced by a dependent module counts as a
533                use as far as kmod is concerned.  */
534             d->flags |= MOD_USED_ONCE;
535         }
536
537         /* Free our temporary memory.  */
538         put_mod_name(n_name);
539         put_mod_name(name);
```

由于 module 结构本身已经复制过来，现在只要复制除此以外的那一部分就够了。对于 i386 处理器，module_arch_init 是空操作，总是返回 0，所以没有作用。对有些处理器，复制过来的内容有一部分可能还在高速缓存中而没有真正写入到内存中，所以要通过 flush_icache_range()将这些内容“冲刷”到内存中去，但是对 i386 处理器而言，这并不是个问题，所以这个函数也是空操作。

前面讲过，模块之间可能有依赖关系，正在安装中的模块可能要引用其它模块中的符号。虽然在用户空间已经完成了对这些符号的连接，但现在必须验证这些模块还在内核中未被拆除。所以，这里通过一个 for 循环检验模块的 deps 数组（见 504 行）。对于数组中的每一个元素，即 module_ref 结构，一方面是检查其 dep 指针（这些指针是在用户空间的连接阶段设置好了的）并不是指向目标模块自身；另一方面是扫描 module_list，如果 dep 指针所指的 module 结构已经不在 module_list 中（见 515 行的 for 循环），那么这次系统调用就失败了，对目标模块的安装也就失败了。在这种情况下，应用程序（如 insmod）有责任通过系统调用 delete_module()将已经创建的 module 结构从 module_list 中删除。

通过了第一趟扫描以后，还要再来一次扫描（见 526 行），这一次要将每个 module_ref 结构链入到所依赖模块（指针 d 指向这个模块的 module 结构）的 refs 队列中，并将结构中的 ref 指针指向正在安装中的 module 结构。这样，每个 module_ref 结构既存在于其所属模块（即正在安装中的模块）的 deps[]数组中，又出现于该模块所依赖的某个模块的 refs 队列中，成为连接两个模块的纽带。从一个模块的 deps[]数组可以找到这个模块所依赖的所有模块，而沿着它的 refs 链则可以找到所有依赖于它的模块，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

从而形成了模块在内核中的“关系网”。

至此，模块的安装已基本完成，为模块安装而分配的临时缓冲区 n_name 和 name 都可以释放了。剩下的大事情还有一件，那就是启动执行模块的 init_module()函数。

==================== kernel/module.c 541 569 ====================

[sys_init_module]

```
	/* Initialize the module.  */
	mod->flags |= MOD_INITIALIZING;
	atomic_set(&mod->uc.usecount,1);
	if (mod->init && (error = mod->init()) != 0) {
		atomic_set(&mod->uc.usecount,0);
		mod->flags &= ~MOD_INITIALIZING;
		if (error > 0)	/* Buggy module */
			error = -EBUSY;
		goto err0;
	}
	atomic_dec(&mod->uc.usecount);

	/* And set it running.  */
	mod->flags = (mod->flags | MOD_RUNNING) & ~MOD_INITIALIZING;
	error = 0;
	goto err0;

err3:
	put_mod_name(n_name);
err2:
	*mod = mod_tmp;
	strcpy((char *)mod->name, name_tmp);	/* We know there is room for this */
err1:
	put_mod_name(name);
err0:
	unlock_kernel();
	kfree(name_tmp);
	return error;
}
```

不提供 init_module()函数，也就是让 module 结构中的函数指针 init 为 NULL 也是允许的，但是那样就使模块的存在失去了意义（除占据了一些内核空间外），因为到这里为止内核还没有任何途径访问模块中的变量或调用模块中的任何函数。所以，在正常情况下都会提供模块的 init_module()函数。如果这个函数正常完成了它的操作就应该返回 0，否则模块的安装就失败了。

读者在后面将会看到一个典型 init_module()函数的代码。

最后一个系统调用是 delete_module()，用来拆卸已安装的模块，其内核中的实现为 sys_delete_module()。我们仍旧分段来读（kernel/module.c）：

==================== kernel/module.c 586 662 ====================

```
asmlinkage long
sys_delete_module(const char *name_user)
```

```
{
	struct module *mod, *next;
	char *name;
	long error;
	int something_changed;

	if (!capable(CAP_SYS_MODULE))
		return -EPERM;

	lock_kernel();
	if (name_user) {
		if ((error = get_mod_name(name_user, &name)) < 0)
			goto out;
		if (error == 0) {
			error = -EINVAL;
			put_mod_name(name);
			goto out;
		}
		error = -ENOENT;
		if ((mod = find_module(name)) == NULL) {
			put_mod_name(name);
			goto out;
		}
		put_mod_name(name);
		error = -EBUSY;
		if (mod->refs != NULL)
			goto out;

		spin_lock(&unload_lock);
		if (!__MOD_IN_USE(mod)) {
			mod->flags |= MOD_DELETED;
			spin_unlock(&unload_lock);
			free_module(mod, 0);
			error = 0;
		} else {
			spin_unlock(&unload_lock);
		}
		goto out;
	}

	/* Do automatic reaping */
restart:
	something_changed = 0;
	for (mod = module_list; mod != &kernel_module; mod = next) {
		next = mod->next;
		spin_lock(&unload_lock);
		if (mod->refs == NULL
		    && (mod->flags & MOD_AUTOCLEAN)
		    && (mod->flags & MOD_RUNNING)
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
           && !(mod->flags & MOD_DELETED)
           && (mod->flags & MOD_USED_ONCE)
           && !__MOD_IN_USE(mod)) {
            if ((mod->flags & MOD_VISITED)
               && !(mod->flags & MOD_JUST_FREED)) {
                spin_unlock(&unload_lock);
                mod->flags &= ~MOD_VISITED;
            } else {
                mod->flags |= MOD_DELETED;
                spin_unlock(&unload_lock);
                free_module(mod, 1);
                something_changed = 1;
            }
        } else {
            spin_unlock(&unload_lock);
        }
    }
    if (something_changed)
        goto restart;
    for (mod = module_list; mod != &kernel_module; mod = mod->next)
        mod->flags &= ~MOD_JUST_FREED;
    error = 0;
out:
    unlock_kernel();
    return error;
}
```

与前几个系统调用一样，只有特权用户方可允许拆卸模块。拆卸模块有两种方式：参数 name_user 为非 0，表示拆除一个特定的模块；为 0 则表示“清理仓库”，即拆除所有可以拆除的模块。我们先看有具体目标的拆除。从用户空间复制模块名之后，就通过 find_module()从 module_list 中找到目标模块的 module 结构。模块的拆卸也是有条件的，如果内核中还有其它模块依赖于目标模块，即引用目标模块中的符号，那就不能把目标模块拆除。怎么知道是否有模块依赖于目标模块呢？看目标模块的 refs 指针是否为非 0 就可以了。及时已经没有模块依赖于目标模块，也还不能够说明就可以立即拆除这个模块，这里还有一个条件，即__MOD_IN_USE(mod)的值为 0，也就是说目标模块不在使用中。宏操作__MOD_IN_USE()定义于 include/linux/module.h：

==================== include/linux/module.h 147 149 ====================

```
#define __MOD_IN_USE(mod)                                      \
    (mod_member_present((mod), can_unload) && (mod)->can_unload  \
     ? (mod)->can_unload() : atomic_read(&(mod)->uc.usecount))
```

从上述宏定义可以看出，如果 module 结构中包括了函数指针 can_unload，并且这个函数指针非 0，就调用这个函数，并根据其返回值来决定是否可以拆除；否则就看 module 结构中的使用计数 uc.usecount。在前面 sys_init_module()的代码中，我们看到计数器 uc.usecount 在调用 init_module()之前设置成 1，而在调用之后，则将其递减成 0（见 543 行和 551 行）。这个计数器为非 0 表示正在对模块进行某种操作，所以此时若要求拆卸便失败而返回出错代码-EBUSY。反之，如果一切顺利，就调用

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

kernel/module.c 中的函数 free_module()实施拆除：

==================== kernel/module.c 1012 1056 ====================

[sys_delete_module()>free_module()]

```
/*
 * Free the given module.
 */

void
free_module(struct module *mod, int tag_freed)
{
	struct module_ref *dep;
	unsigned i;

	/* Let the module clean up.  */

	if (mod->flags & MOD_RUNNING)
	{
		if(mod->cleanup)
			mod->cleanup();
		mod->flags &= ~MOD_RUNNING;
	}

	/* Remove the module from the dependency lists.  */

	for (i = 0, dep = mod->deps; i < mod->ndeps; ++i, ++dep) {
		struct module_ref **pp;
		for (pp = &dep->dep->refs; *pp != dep; pp = &(*pp)->next_ref)
			continue;
		*pp = dep->next_ref;
		if (tag_freed && dep->dep->refs == NULL)
			dep->dep->flags |= MOD_JUST_FREED;
	}

	/* And from the main module list.  */

	if (mod == module_list) {
		module_list = mod->next;
	} else {
		struct module *p;
		for (p = module_list; p->next != mod; p = p->next)
			continue;
		p->next = mod->next;
	}

	/* And free the memory.  */

	module_unmap(mod);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

与安装模块时调用 init_module()相对应，拆除模块时首先要调用目标模块的 cleanup_module()函数（1027 行）。通常，在这个函数中要将模块在系统中的“登记”撤销，使系统不能再看到这个模块的存在。读者以后会看到典型的 cleanup_module()函数代码。调用了这个函数以后，就失去了进入这个模块的途径，从而在概念上这个模块已经不在运行了，所以把模块的 MOD_RUNNING 标志位清 0。

拆除一个模块以后，它所依赖的所有其它模块就都少了一个“用户”。如前所述，每个模块都有一个 refs 队列，它的所有“用户”模块都通过一个 module_ref 数据结构链入到这个队列中。所以，现在要把目标模块从它所依赖的所有模块的 refs 队列中脱链。这里的 for 循环（1033 行），是检查目标模块的 deps 数组，依次处理数组中的每个 module_ref 结构，结构中的指针 dep 指向所依赖的模块（它的 module 结构）。所以，dep->dep 指向其中某个模块的 module 结构，而 dep->dep->refs 进而指向其 refs 队列。第二层 for 循环（1035 行）顺着这个队列找到指向上述 module_ref 结构的指针，使 pp 指向这个指针。由于 module_ref 结构中的指针 next_ref 指向该队列中的下一个 module_ref 结构，1037 行就使属于目标模块的 module_ref 结构从队列中脱链。这里之所以需要第二个 for 循环是因为 refs 队列是单链队列，只有顺着这个队列才能找到需要脱链的数据结构。从队列中脱链的 module_ref 结构并不需要（也不应该）单独释放，因为整个 deps 数组都是随模块的 module 结构（以及模块映象的空间）一起作为一个整体分配的。所以，最后将目标模块的 module 结构也从 module_list 队列中脱链以后，就通过 module_unmap()将模块所占用的所有内存资源作为一个整体释放（1055 行）。

一个模块的拆除有可能使它所依赖的其它模块得到“自由”。如果将一个 module_ref 结构从一个模块的 refs 队列脱链以后使这个队列变成了空队列，那么这个模块就再没有其它模块依赖于它了。在这种情况下，如果参数 tag_freed 为非 0，就要设置这个模块的 MOD_JUST_FREED 标志位，表示这个模块刚刚得到自由。不过，在 sys_delete_module()中有目标地拆除个别模块而调用 free_module()时这个参数为 0（见 620 行）。

回到 sys_delete_module()的代码中，如果调用时的参数 name_user 为 NULL，就表示要拆除内核中所有可以拆除的模块。另一方面，在有目标地拆除了一个模块之后，由于可能有模块得到了自由，也要进一步拆除所有可以拆除的模块。所谓“可以拆除”，是指同时满足以下条件：

(1) 不再有任何模块依赖于它（634 行）。
(2) 安装时带有 MOD_AUTOCLEAN 标志位，允许自动拆除（635 行）。
(3) 处于运行状态，即已经安装完毕但尚未拆除（636 行）。
(4) 尚未开始拆除（637 行），注意在上面的 618 行已经设置了模块的 MOD_DELETED 标志，然后在解了锁以后才调用 free_module()，而 MOD_RUNNING 标志位则是在 free_module()里面在调用了模块的 cleanup_module()函数以后才清 0 的。所以 MOD_DELETED 标志表示拆除的过程已经启动。
(5) 模块安装以后已经受到过引用（638 行）。这种引用既可能来自其它模块，也可能来自内核。例如，在安装一个文件系统（指设备）时，如果所涉及的文件系统（指格式）是以模块方式实现的，则就要对此模块的 module 结构调用一个函数 try_inc_mod_count()，递增其使用计数并设置其 MOD_USED_ONCE 标志位。
(6) 模块已不在使用中（639 行）。

代码中通过一个 for 循环扫描 module_list 队列，寻找同时符合所有这些条件的模块。这样的扫描可能要反复多次，因为每拆除一个模块就有可能解放出其它一些模块，而使其中的某些模块也满足了

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

自动拆除的条件。所以，每拆除一个模块以后就把 something_changed 置成 1，使得在 for 循环结束以后再转回 restart 标号处（629 行）开始对 module_list 新一轮的扫描。

特权用户可以执行实用程序/sbin/insmod 和/sbin/rmmod，通过内核提供的上述 4 个系统调用完成具体模块的安装和拆卸。由于/sbin/insmod 实际上还是个相当复杂的目标代码连接过程，用户开发的程序中一般不应该、也不必要自行直接调用 create_module()、init_module()等系统调用。

在由用户通出/sbin/insmod 安装模块的过程中，内核处于一种被动的地位。但是，在很多情况下内核需要主动地启动某个模块的安装，而不能只是消极地等待。例如，读者在第 4 章关于系统调用 exec()的代码中曾经看到，当内核打开一个特殊格式的二进制可执行程序，却发现内核中并没有支持这种格式的目标代码装入程序时，就会试图先装入支持这种格式的可安装模块。具体的代码在 exec.c 中的 search_binary_handler()中，读者不妨回过去看一下。类似的情况还有很多，例如当内核从网络中接收到一个特殊的 packet 或报文，而支持相应规程的模块尚未安装。又如当内核检测到某种硬件，而支持这种硬件的模块尚未安装，等等。

在这样的情况下，内核都通过一个函数 request_module()主动地启动模块的安装。所以，许多模块的安装实际上都是在用户不知不觉中由内核自行启动/sbin/insmod 安装的。让我们来看看这个自动安装的过程。函数 request_module()的代码在 kernel/kmod.c 中：

==================== kernel/kmod.c 159 239 ====================

```
/**
 *   request_module - try to load a kernel module
 *   @module_name: Name of module
 *
 *   Load a module using the user mode module loader. The function returns
 *   zero on success or a negative errno code on failure. Note that a
 *   successful module load does not mean the module did not then unload
 *   and exit on an error of its own. Callers must check that the service
 *   they requested is now available not blindly invoke it.
 *
 *   If module auto-loading support is disabled then this function
 *   becomes a no-operation.
 */

int request_module(const char * module_name)
{
     pid_t pid;
     int waitpid_result;
     sigset_t tmpsig;
     int i;
     static atomic_t kmod_concurrent = ATOMIC_INIT(0);
#define MAX_KMOD_CONCURRENT 50   /* Completely arbitrary value - KAO */
     static int kmod_loop_msg;

     /* Don't allow request_module() before the root fs is mounted!  */
     if ( ! current->fs->root ) {
          printk(KERN_ERR "request_module[%s]: Root fs not mounted\n",
               module_name);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                return -EPERM;
        }

        /* If modprobe needs a service that is in a module, we get a recursive
         * loop.  Limit the number of running kmod threads to max_threads/2 or
         * MAX_KMOD_CONCURRENT, whichever is the smaller.  A cleaner method
         * would be to run the parents of this process, counting how many times
         * kmod was invoked.  That would mean accessing the internals of the
         * process tables to get the command line, proc_pid_cmdline is static
         * and it is not worth changing the proc code just to handle this case.
         * KAO.
         */
        i = max_threads/2;
        if (i > MAX_KMOD_CONCURRENT)
                i = MAX_KMOD_CONCURRENT;
        atomic_inc(&kmod_concurrent);
        if (atomic_read(&kmod_concurrent) > i) {
                if (kmod_loop_msg++ < 5)
                        printk(KERN_ERR
                               "kmod: runaway modprobe loop assumed and stopped\n");
                atomic_dec(&kmod_concurrent);
                return -ENOMEM;
        }

        pid = kernel_thread(exec_modprobe, (void*) module_name, 0);
        if (pid < 0) {
                printk(KERN_ERR "request_module[%s]: fork failed, errno %d\n", module_name, -pid);
                atomic_dec(&kmod_concurrent);
                return pid;
        }

        /* Block everything but SIGKILL/SIGSTOP */
        spin_lock_irq(&current->sigmask_lock);
        tmpsig = current->blocked;
        siginitsetinv(&current->blocked, sigmask(SIGKILL) | sigmask(SIGSTOP));
        recalc_sigpending(current);
        spin_unlock_irq(&current->sigmask_lock);

        waitpid_result = waitpid(pid, NULL, __WCLONE);
        atomic_dec(&kmod_concurrent);

        /* Allow signals again.. */
        spin_lock_irq(&current->sigmask_lock);
        current->blocked = tmpsig;
        recalc_sigpending(current);
        spin_unlock_irq(&current->sigmask_lock);

        if (waitpid_result != pid) {
                printk(KERN_ERR "request_module[%s]: waitpid(%d,...) failed, errno %d\n",
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                module_name, pid, -waitpid_result);
    }
    return 0;
}
```

首先检查文件系统的根设备是否已经安装。由内核启动的模块安装也要靠/sbin/insmod 来完成，因此它在根设备尚未安装前显然是无法进行的。注意，request_module()是在当前进程的上下文中（而不是作为中断服务程序）执行的，这样才能使当前进程进入睡眠而等待模块安装的完成。对 request_module()的调用有可能会嵌套，因为在安装的过程中可能会发现必须先安装另一个模块。因此，就要对嵌套深度加以限制，程序中设置了一个静态变量 kmod_concurrent，作为嵌套深度的计数器，并且还规定了嵌套深度的上限 MAX_KMOD_CONCURRENT。不过，对嵌套深度的控制还要考虑到系统中对进程数量的限制，即 max_threads，因为在安装的过程中要创建临时的进程。

通过了这些检查以后，就调用 kernel_thread()创建一个内核线程 exec_modprobe()。对于 kernel_thread()本身，读者已经在第 4 章中看到过了，所以我们继续往下先把 request_module()代码的剩余部分读完，然后再回过来看 exec_modprobe()的代码。

创建内核线程成功以后，先把当前进程的信号机制中除 SIGKILL 和 SIGSTOP 以外所有的信号都屏蔽掉，免得在等待模块安装的过程中受到干扰，然后就通过 waitpid()使当前进程进入睡眠，静候佳音，等待创建出来的内核线程在完成模块安装以后退出舞台。当前进程被唤醒而从 waitpid()返回时，内核线程 exec_modprobe()的运行已经结束，恢复了当前进程原有的信号设置以后，根据返回值可以判断 exec_modprobe()的操作是否成功。如果失败，就通过 printk()在系统的“运行日志”/var/log/messages 中输出一条出错信息。

好，现在我们来看 exec_modprobe()的代码。它还是在 kernel/kmod.c 中：

==================== kernel/kmod.c 139 157 ====================

```
/*
    modprobe_path is set via /proc/sys.
*/
char modprobe_path[256] = "/sbin/modprobe";

static int exec_modprobe(void * module_name)
{
    static char * envp[] = { "HOME=/", "TERM=linux", "PATH=/sbin:/usr/sbin:/bin:/usr/bin", NULL };
    char *argv[] = { modprobe_path, "-s", "-k", "--", (char*)module_name, NULL };
    int ret;

    ret = exec_usermodehelper(modprobe_path, argv, envp);
    if (ret) {
        printk(KERN_ERR
                "kmod: failed to exec %s -s -k %s, errno = %d\n",
                modprobe_path, (char*) module_name, errno);
    }
    return ret;
}
```

这里的 modprobe_path 就是路径名/sbin/modprobe，所以，如果 module_name 为 mymodule 的话，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的 argv[]就相应于命令：

/sbin/modprobe -s -k mymodule

选择项-s 表示在安装过程中产生的信息应写入系统的运行日志，而不要在控制终端上显示；-k 表示安装时将模块的 MOD_AUTOCLEAN 标志位设成 1。

函数 exec_usermodehelper()的代码也在同一文件（kernel/kmod.c）中：

==================== kernel/kmod.c 86 135 ====================

[exec_modprobe()>exec_usermodehelper()]

```
int exec_usermodehelper(char *program_path, char *argv[], char *envp[])
{
	int i;
	struct task_struct *curtask = current;

	curtask->session = 1;
	curtask->pgrp = 1;

	use_init_fs_context();

	/* Prevent parent user process from sending signals to child.
	   Otherwise, if the modprobe program does not exist, it might
	   be possible to get a user defined signal handler to execute
	   as the super user right after the execve fails if you time
	   the signal just right.
	*/
	spin_lock_irq(&curtask->sigmask_lock);
	sigemptyset(&curtask->blocked);
	flush_signals(curtask);
	flush_signal_handlers(curtask);
	recalc_sigpending(curtask);
	spin_unlock_irq(&curtask->sigmask_lock);

	for (i = 0; i < curtask->files->max_fds; i++ ) {
		if (curtask->files->fd[i]) close(i);
	}

	/* Drop the "current user" thing */
	{
		struct user_struct *user = curtask->user;
		curtask->user = INIT_USER;
		atomic_inc(&INIT_USER->__count);
		atomic_inc(&INIT_USER->processes);
		atomic_dec(&user->processes);
		free_uid(user);
	}

	/* Give kmod all effective privileges.. */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    curtask->euid = curtask->fsuid = 0;
    curtask->egid = curtask->fsgid = 0;
    cap_set_full(curtask->cap_effective);

    /* Allow execve args to be in kernel space. */
    set_fs(KERNEL_DS);

    /* Go, go, go... */
    if (execve(program_path, argv, envp) < 0)
        return -errno;
    return 0;
}
```

这个函数是在内核线程 exec_modprobe()的上下文中运行的，所以这里的 current 指向这个内核线程的 task_struct 结构，而与创建这个线程时的 current 不同，那时候的 current 指向当时的当前进程，即 exec_modprobe()的父进程。内核线程 exec_modprobe()从其父进程那里继承了绝大部分资源和特性，包括它的 fs_struct 的内容和所有的已打开文件，以及它的进程号、组号，还有所有的特权。但是，这些从父进程继承下来的资源和特性未必满足模块安装的要求。首先是父进程的根目录，在“文件系统”一章中讲过，一个进程可以设置自身的根目录，所以父进程的根目录有可能并不是真正的整个文件系统的根目录。如果那样的话，在父进程的根目录中也许根本就没有 sbin 这么个子目录，更找不到 modprobe 了。所以，首先得要确保这个内核线程的 fs_srtuct 中的指针 root 确实指向文件系统的总根，结构中的其它一些内容也要与其相一致。怎么来保证这一点呢？系统中只有一个进程，即 init_task 的根目录是肯定不会改变的，所以这里通过 kernel/kmod.c 中的一个函数 use_init_fs_context()从 init_task 结构中直接继承其根目录等资源。

==================== kernel/kmod.c 32 84 ====================

[exec_modprobe()>exec_usermodehelper()>use_init_fs_context()]

```
static inline void
use_init_fs_context(void)
{
    struct fs_struct *our_fs, *init_fs;
    struct dentry *root, *pwd;
    struct vfsmount *rootmnt, *pwdmnt;

    /*
     * Make modprobe's fs context be a copy of init's.
     *
     * We cannot use the user's fs context, because it
     * may have a different root than init.
     * Since init was created with CLONE_FS, we can grab
     * its fs context from "init_task".
     *
     * The fs context has to be a copy. If it is shared
     * with init, then any chdir() call in modprobe will
     * also affect init and the other threads sharing
     * init_task's fs context.
```

```
51          *
52          * We created the exec_modprobe thread without CLONE_FS,
53          * so we can update the fields in our fs context freely.
54          */
55
56         init_fs = init_task.fs;
57         read_lock(&init_fs->lock);
58         rootmnt = mntget(init_fs->rootmnt);
59         root = dget(init_fs->root);
60         pwdmnt = mntget(init_fs->pwdmnt);
61         pwd = dget(init_fs->pwd);
62         read_unlock(&init_fs->lock);
63
64         /* FIXME - unsafe ->fs access */
65         our_fs = current->fs;
66         our_fs->umask = init_fs->umask;
67         set_fs_root(our_fs, rootmnt, root);
68         set_fs_pwd(our_fs, pwdmnt, pwd);
69         write_lock(&our_fs->lock);
70         if (our_fs->altroot) {
71              struct vfsmount *mnt = our_fs->altrootmnt;
72              struct dentry *dentry = our_fs->altroot;
73              our_fs->altrootmnt = NULL;
74              our_fs->altroot = NULL;
75              write_unlock(&our_fs->lock);
76              dput(dentry);
77              mntput(mnt);
78         } else
79              write_unlock(&our_fs->lock);
80         dput(root);
81         mntput(rootmnt);
82         dput(pwd);
83         mntput(pwdmnt);
84    }
```

对这一段代码就不需要多说了（读者若感到困难就应该复习一下第 5 章中的有关内容）。解决了根目录的问题以后，还要把从父进程继承的待处理信号全都“冲刷”掉，即全部丢弃，并且把父进程所设置的信号处理也全部恢复成由系统默认的处理方式。然后，继承下来的全部已打开文件也要关闭，连打开文件号 0、1、2 所代表的三个标准输入 / 输出和出错输出文件也都关闭，这样就与父进程的控制终端脱离了关系。不但如此，连线程的 uid、euid、fsuid 也要全部换成 0，使其变成特权（超级）用户进程，还要通过 cap_set_full()赋予其全部特权（第 126 行）。读者也许感到迷惑，怎么这个进程有这么大的能耐，竟然可以为所欲为，不受限制？原因就在于这是个内核线程，而不是进程。

第 129 行的 set_fs()是个宏操作，定义于 include/asm-i386/uaccess.h：

==================== include/asm-i386/uaccess.h 30 30 ====================

```
30    #define set_fs(x)  (current->addr_limit = (x))
```

也就是说，把从父进程继承的地址上限也换成 KERNEL_DS，即 0xffffffff（4GB），因为父进程的地址上限很可能是 USER_DS，即 0xbfffffff（3GB）。

经过这么一番脱胎换骨的改造，内核线程 exec_modprobe()就成了一个彻底的特权用户进程(线程)，而与原先的父进程没有多大联系了。套用一句时下流行的话，这叫“借壳上市”。但是，两个进程（线程）间的父子关系还在，当 exec_modproc()死亡退出运行时还是会唤醒其父进程。

最后，就是通过 execve()执行/sbin/modprobe 了，有关 ececve()的详情，包括运行结束时的处理，可参阅第 4 章。

在阅读模块的 init_module()和 cleanup_module()两个函数之前，还要讲一下内核中符号的“移出 ”问题，也就是内核中的哪一些符号可以被可安装模块引用。内核中的每一个符号都必须通过宏定义 EXPORT_SYMBOL 明确规定准予移出，才能由/sbin/insmod 通过 query_module()系统调用获得这些符号的地址，否则就都是不准移出的。下面是取自 kernel/ksyms.c 中的几个例子：

==================== kernel/ksyms.c 142 148 ====================

```
EXPORT_SYMBOL(path_init);
EXPORT_SYMBOL(path_walk);
EXPORT_SYMBOL(path_release);
EXPORT_SYMBOL(__user_walk);
EXPORT_SYMBOL(lookup_one);
EXPORT_SYMBOL(lookup_hash);
EXPORT_SYMBOL(sys_close);
```

宏定义 EXPORT_SYMBOL 将给定符号的符号名和它的值，即该符号在连接后的内核映象中的地址、组装在一个 module_symbol 数据结构中，并指示连接程序（ld）在连接内核映象时将这个结构放在一个称为“ __ksymtab”的区段中。这样，连接以后所有这样的数据结构就都在__ksymtab 区段中，而这个区段就成了内核对外公开的符号表。而“对内”的符号表则由连接程序自行生成，并仅供连接程序 ld 自己使用。数据结构类型 module_symbol 是在 include/linux/module.h 中定义的：

==================== include/linux/module.h 37 41 ====================

```
struct module_symbol
{
	unsigned long value;
	const char *name;
};
```

这里的 name 只是一个字符指针，真正的字符则存放在另一个区段“ .kstrtab”中。这样做的好处在于使符号表的结构变得规则，因为不管符号名字符串有多长，指针的大小总是固定的。

这里还要指出，内核对可安装模块的支持是可选的。如果在编译内核代码之前的系统配置（config）阶段选择了支持可安装模块，就定义了编译提示 CONFIG_MODULES，使支持可安装模块的代码受到编译，而 EXPORT_SYMBOL 也只有在这种情况下才有定义。

与 EXPORT_SYMBOL 有关的定义都在 include/linux/module.h 中：

==================== include/linux/module.h 151 154 ====================

```
/* Indirect stringification.  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
152
153 #define __MODULE_STRING_1(x)     #x
154 #define __MODULE_STRING(x)   __MODULE_STRING_1(x)
```

===================== include/linux/module.h 325 336 =====================

```
325 #define __EXPORT_SYMBOL(sym, str)             \
326 const char __kstrtab_##sym[]                  \
327 __attribute__((section(".kstrtab"))) = str;       \
328 const struct module_symbol __ksymtab_##sym        \
329 __attribute__((section("__ksymtab"))) =           \
330 { (unsigned long)&sym, __kstrtab_##sym }
331
332 #if defined(MODVERSIONS) || !defined(CONFIG_MODVERSIONS)
333 #define EXPORT_SYMBOL(var)  __EXPORT_SYMBOL(var, __MODULE_STRING(var))
334 #else
335 #define EXPORT_SYMBOL(var)  __EXPORT_SYMBOL(var, __MODULE_STRING(__VERSIONED_SYMBOL(var)))
336 #endif
```

第 332～336 行是条件编译，即在不同条件下对 EXPORT_SYMBOL 有不同的定义。为什么要有不同呢？这是为了保证内核与可安装模块在版本上的严格一致。要保证二者的版本严格一致，最好的办法就是将版本信息编码进变量名中，例如将版本号作为变量名的后缀。这样，如果变量相同而版本号不一致，/sbin/insmod 就会认为是两个不同的符号而不予连接。但是，另一方面，这在一定程度上也带来了不便。因为每当有版本变动时就要重新编译（或从网上下载）许多可安装模块，而有些版本变动实际上只是很小的变动，并不影响运行。因此，内核把是否将版本信息编码进符号名作为一个可选项提供。如果需要那样做的话，就可以在编译内核代码前的系统配置阶段选择这个可选项，而使条件编译提示 CONHG_MODVERSIONS 成为有定义。有时候，虽然从总体上选择了使用版本信息，但对于某些源代码和模块中的符号却并不需要搞得这么复杂，这时就可以在编译这些源代码时在命令行中加上“ -D MODVERSIONS”可选项，或在源代码文件中加上“ #define MODVERSIONS”，使该文件中定义的符号摆脱可选项 CONFIG_MODVERSIONS 的控制。

我们先看不带版本信息的符号表项是如何建立的。此时的宏操作 EXPORT_SYMBOL 定义于第 333 行。以符号 path_init 为例，此时 EXPORT_SYMBOL(path_init) 的定义就成了__EXPORT_SYMBOL(path_init, "path_init")。而第 325 行又进而将其定义为两个语句，第一个语句（326～327 行）定义了一个名为__kstrtab_path_init 的字符串，将字符串的内容初始化为“ path_init”，并将其置于内核映象中的.kstrtab 区段。第二个语句(328～330 行)则定义了一个名为__ksymtab_path_init 的 module_symbol 数据结构，将其初始化成{&path_init, __kstrtab_path_init}，并将其置于内核映象中的_ksymtab 区段。这样，这个数据结构中的字段 value 的值为 path_init 在内存映象中的地址，而指针 name 则指向字符串“ path_init”。顺便提一下，这里定义的两项数据都是 const，初始化以后便不容改变。

采用版本信息时的操作就要复杂一些了。不同之处在于__EXPORT_SYMBOL 的第二个参数先要经过另一个宏操作__VERSIONED_SYMBOL 的变换，这个宏操作定义于 include/linux/modsetver.h：

===================== include/linux/modsetver.h 3 6 =====================

```
#define __SYMBOL_VERSION(x)       __ver_ ## x
#define __VERSIONED_SYMBOL2(x,v)  x ## _R ## v
#define __VERSIONED_SYMBOL1(x,v)  __VERSIONED_SYMBOL2(x,v)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
6    #define __VERSIONED_SYMBOL(x)    __VERSIONED_SYMBOL1(x,__SYMBOL_VERSION(x))
```

仍以 path_init 为例，宏操作__SYMBOL_VERSION(path_init)首先将其变换成__ver_path_init；然后__VERSIONED1 又将 path_init、_R 以及__ver_path_init 三部分连在一起。可是，经过这样的处理，最后形成的字符串中并没有版本信息的编码啊！奥妙就在于后缀__ver_path_init 并不是最终用来与前面两部分相连的字符串，它也还只是一个中间产物。在某个源代码文件中还会有类似于“#define __ver_path_init smp_1234abcd”这样的宏定义。这样，把三部分连在一起以后就形成了类似于“path_init_Rsmp_1234abcd”这样的字符串（这里 smp 表示内核支持 SMP 多处理器结构），最后字符串__kstrtab_path_init 的内容就成为“path_init_Rsmp_1234abcd”，而数据结构__ksymtab_path_init 中的内容则仍为{&path_init, __kstrtab_path_init}。显然，_Rsmp_1234abcd 即为包含着版本信息的符号名后缀。那么，这个后缀是怎么来的呢？这是由一个工具/sbin/genksyms 根据版本号和具体符号的类型编码（CRC）产生的。熟悉 C++的读者一定会联想起 C++中对符号（变量名或函数名）的编码（称为 mangling）。读者可以在内核的 Makefile 和 Rules.make 中找到对这个工具的应用，还可以用命令“man genksyms”看一下这个工具的说明。由工具产生的输出写入到一些.ver 文件中，这些.ver 文件则又被 include 到 include/linux/modversions.h 中。这个.h 文件是在通过 make 生成内核映象的过程中生成的，看一下它的具体内容，我们就可以发现大量的类似于“#include <linux/ksyms.ver>”这样的“语句”。

只要内核映象中的__ksymtab 区段非空，内核就有对外公开的符号，可以通过系统调用 query_module()查询。此外，在/proc 目录中有个特殊文件 ksyms，用来提供内核中的“移出”符号清单（包括已安装模块的符号）。读者不妨用命令“more /proc/ksyms”看一下你的内核符号是否包含版本信息编码。

现在，我们可以来看几个典型模块的 init_module()和 cleanup_module()函数了。

我们要看的第一个模块是一个声卡驱动程序。名为“sparcaudio”。我们的目的并不在于了解具体的驱动程序代码，而主要在于了解它在 init_module()中向系统登记、从而建立起从内核中文件系统的 vfs 层进入其具体驱动程序的途径这么一个过程，以及在 cleanup_module()中将这个途径撤销的过程。另一个使我们对它感兴趣的原因是，它还并不是真正直接驱动声卡硬件的模块，而只是 vfs 层与物理层之间的一个中间层次。它一方面向系统登记，建立起从 vfs 层进入它的途径；另一方面又准备接受来自更低层模块的登记，从而建立起进入物理层的途径。像这样由不同的模块来实现一种系统结构中的不同层次，从而形成一个模块“堆栈”的做法，是一种典型的程序结构和实现方式（尤其是在计算机网络环境下）。同时，它也是模块间依赖关系的一个实例。这个模块的代码在 drivers/sbus/audio/audio.c 中。路径名中的 drivers/sbus 表示有关的代码都是 SUN 公司的 Sparc 结构工作站中 sbus 总线设备的驱动程序。我们不准备深入到 sbus 的细节中去，而只是用它作为一个可安装模块的实例。

对 sparcaudio 的支持既可以编译成一个可安装模块，也可以静态地编译连接进内核的映象。内核编译之前的系统配置阶段为用户提供了选择手段。通常，对每一种设备或功能都有一个问题让用户回答，这个问题就是要或者不要、以及以何种方式支持某种设备或功能。如果回答“Y”就表示要，并且将相应的代码静态地连接在内核映象中；回答“M”表示要，但是将代码编译成可安装模块；否则就回答“N”，表示不要。当选择将代码编译成可安装模块时，模块的入口就是 init_module()，就好像每个可执行程序的入口都是 main()一样。虽然每个模块都有一个 init_module()，但是这些模块不会由 ld

PDF-XCHANGE Click to buy NOW! www.docu-track.com

连接在一起，所以不会互相冲突。但是，如果将代码编译并连接进内核映象，那就不能使用 init_module()这个函数名了，那样会造成符号名冲突而不能连接。所以，对 sparcaudio 不采用可安装模块方式时，就要给 init_module()函数换个名字，这里叫 sparcaudio_init()，并且加上了说明__init，表示这个函数是个初始化函数，同时还要在代码中加上这么一行：

==================== drivers/sbus/audio/audio.c 2230 2230 ====================

```
2230 module_init(sparcaudio_init)
```

表示在系统初始化时应调用这个函数。从这里也可以看出，即使不把它编译成可安装模块，代码的设计和实现仍然是高度模块化的。

先看 sparcaudio_init()，这个函数在 drivers/sbus/audio/audio.c 中：

drivers/sbus/audio/audio.c

```
2200 static int __init sparcaudio_init(void)
2201 {
2202     /* Register our character device driver with the VFS. */
2203     if (devfs_register_chrdev(SOUND_MAJOR, "sparcaudio", &sparcaudio_fops))
2204         return -EIO;
2205 
2206     devfs_handle = devfs_mk_dir (NULL, "sound", NULL);
2207 
2208 #ifdef CONFIG_SPARCAUDIO_AMD7930
2209     amd7930_init();
2210 #endif
2211 #ifdef CONFIG_SPARCAUDIO_DBRI
2212       dbri_init();
2213 #endif
2214 #ifdef CONFIG_SPARCAUDIO_CS4231
2215     cs4231_init();
2216 #endif
2217 #ifdef CONFIG_SPARCAUDIO_DUMMY
2218     dummy_init();
2219 #endif
2220 
2221     return 0;
2222 }
```

前面提到过，当以不同的模块实现一个系统结构中的不同层次时，会形成一个“模块堆栈”。之所以称之为“堆栈”，一方面是因为这是一组模块，另一方面是这些模块也遵循“后进先出”的原则，位于堆栈（层次意义上的）顶部的模块其实就是内核本身，虽然它并不是一个“可安装模块”。从这个意义上说，每一个模块实际上都存在于一个模块堆栈中。我们来考察一下模块堆栈中相邻两层间的界面。首先，位于上层的模块肯定要调用下层模块中的函数。这个函数调用的界面是标准化的，那就是类似于 file_operations 一类包含着函数指针的数据结构，由下层模块通过向其上层“登记”的方式递交给上层。除此以外，上层模块就不能、也不应直接调用下层模块中的函数，或访问下层模块中的变量了。也就是说，上层模块不直接引用下层模块的符号，对下层模块没有依赖关系。可是，反过来下层模块对于上层模块就有依赖关系了。每一个模块都不能脱离它的环境而存在，而运行。这个环境就是由位

PDF-XCHANGE Click to buy NOW! www.docu-track.com

于其上层的模块提供的。至少，它的直接上层应提供一个函数让它可以向上层登记，所以至少要向它移出一个符号。这样，每个模块对位于其上层的模块就有了依赖关系，而除最底层以外的模块就有必要向位于其下层的模块移出若干符号。那么，是否每个模块都直接依赖于位于其上层的所有模块呢？那倒不一定，但是一般对“顶头上司”和位于最高层的内核都有直接的依赖关系，也就是需要直接引用这些模块中的符号。由于 sparcaudio 并不是处于最底层的模块，它也要向它下层的模块移出一些符号。对于 2.1.0 以前的版本，模块要通过一个函数 register_symtab()向内核登记它的符号表，而在从那以后的版本中，对模块符号表的处理在形式上就与内核一致了。模块 sparcandio 的符号表定义为：

==================== drivers/sbus/audio/audio.c 2195 2198 ====================

```
EXPORT_SYMBOL(register_sparcaudio_driver);
EXPORT_SYMBOL(unregister_sparcaudio_driver);
EXPORT_SYMBOL(sparcaudio_output_done);
EXPORT_SYMBOL(sparcaudio_input_done);
```

这个模块向低层模块移出的符号有 4 个，都是供位于其下层的模块调用的函数（从符号表中并不能看出具体的符号是函数还是变量，这是由模块间的界面决定的），其中前两个显然是供下层模块登记和撤销登记时调用的函数。后两个则从函数名可以看出是供低层模块在完成输入 / 输出后“回叫”的函数。

向上层的登记是通过 devfs_register_chrdev()进行的，这个函数由内核提供并移出，定义于 fs/devfs/base.c：

==================== fs/devfs/base.c 1935 1940 ====================

[sparcaudio_init()>devfs_register_chrdev()]

```
int devfs_register_chrdev (unsigned int major, const char *name,
                           struct file_operations *fops)
{
    if (boot_options & OPTION_ONLY) return 0;
    return register_chrdev (major, name, fops);
}   /*  End Function devfs_register_chrdev  */
```

登记时使用了三个参数，第一个参数是模块所代表设备的主设备号 SOUND_MAJOR，在 include/linux/major.h 中定义为：

==================== include/linux/major.h 38 38 ====================

```
#define SOUND_MAJOR     14
```

第二个参数为模块名，而第三个参数就是指向该种设备的 file_operations 数据结构的指针，这个数据结构也是在 drivers/sbus/audio/audio.c 中定义的：

==================== drivers/sbus/audio/audio.c 1961 1970 ====================

```
static struct file_operations sparcaudio_fops = {
        owner:          THIS_MODULE,
        llseek:         sparcaudio_lseek,
        read:           sparcaudio_read,
        write:          sparcaudio_write,
        poll:           sparcaudio_select,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    ioctl:        sparcaudio_ioctl,
    open:         sparcaudio_open,
    release:  sparcaudio_release,
};
```

登记的是 file_operations 结构，这也说明在这个模块的上面就是内核（严格说是内核的 vfs 层），而没有其它模块了。另一方面，从函数名 devfs_register_chrdev()也可以看出，这个模块所支持的是树状设备文件系统 devfs，并且声卡设备是字符设备。不过，devfs 的字符设备登记与普通设备文件实际上相同，函数 register_chrdev()的代码在 fs/devices.c 中：

==================== fs/devices.c 98 124 ====================

[sparcaudio_init()>devfs_register_chrdev()>register_chrdev()]

```
int register_chrdev(unsigned int major, const char * name, struct file_operations *fops)
{
    if (major == 0) {
        write_lock(&chrdevs_lock);
        for (major = MAX_CHRDEV-1; major > 0; major--) {
            if (chrdevs[major].fops == NULL) {
                chrdevs[major].name = name;
                chrdevs[major].fops = fops;
                write_unlock(&chrdevs_lock);
                return major;
            }
        }
        write_unlock(&chrdevs_lock);
        return -EBUSY;
    }
    if (major >= MAX_CHRDEV)
        return -EINVAL;
    write_lock(&chrdevs_lock);
    if (chrdevs[major].fops && chrdevs[major].fops != fops) {
        write_unlock(&chrdevs_lock);
        return -EBUSY;
    }
    chrdevs[major].name = name;
    chrdevs[major].fops = fops;
    write_unlock(&chrdevs_lock);
    return 0;
}
```

登记字符设备（以及块设备）时，如果以 0 作为主设备号，就表示要求由内核分配一个主设备号。当然，由内核分配的主设备号只是临时的。内核中有个 device_struct 结构数组 chrdevs[]，这是一个以字符设备的主设备号为下标的数组，每个元素都是一个 device_struct 数据结构，有关的定义在 fs/devices.c 中：

==================== fs/devices.c 33 36 ====================

```
struct device_struct {
    const char * name;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        struct file_operations * fops;
    };
    . . . . . .
```

==================== fs/devices.c 39 40 ====================

```
    static struct device_struct chrdevs[MAX_CHRDEV];

```

所谓登记，就是将由模块提供的 file_operations 结构指针填入这个数组（或称字符设备表）的某个表项。登记以后，只要知道了字符设备的主设备号，就可以很快找到它的 file_operations 数据结构，进而找到该种设备的各种驱动函数。

登记了以后，位于上层的模块（在这里是内核）就可以“看见”这个模块了。但是，应用程序却还不能“看见”它，因而还不能通过系统调用来使用它。要使应用程序能“看见”这个模块或者这个模块所驱动的设备，就要在文件系统中为其创建一个代表它的节点。这个节点可以是在单层的/dev 目录中的一个节点，也可以是在多层的设备文件目录下的一个节点。在单层的/dev 目录中，每个节点都是文件节点，每个节点都代表一个具体的设备，所以要有主设备号和次设备号两项参数才能创建一个节点。这一点读者在 mknod()的代码中已经看到过了。对于多层的 devfs，应给（由主设备号确定的）每一种设备都创建一个目录节点，在这个目录下才是代表具体设备的文件节点。函数 devfs_mk_dir()的作用就是创建这样的目录节点。

调用这个函数时的第一个参数 dir 是一个指针，指向代表父目录节点的 devfs_entry 数据结构，当 dir 为 NULL 时表示父目录节点为 devfs 的根节点，即“ /dev”。与普通目录节点的 dentry 数据结构不同，devfs 树中目录节点的数据结构为 devfs_entry，而指向这种数据结构的指针类型则定义为 devfs_handle_t。第二个参数为待创建目录节点的节点名，这里是“ sound”。第三个参数则为节点名的长度，以 0 为参数表示由 devfs_mk_dir()计算出字符串的长度。最后一个参数为指向附加信息的指针，这里是 NULL，表示并无附加信息。由于读者已经阅读过 path_walk()、mknod()等函数的代码，这里我们就不讲解 devfs_mk_dir()的代码了，有兴趣的读者可自行阅读。

至此，模块 sparcaudio 的初始化实际上已经完成了。但是，这个模块并不是最底层的模块，所以还要考虑它的（直接）下层。如果它的下层也是通过可安装模块实现的，那这里就不用做什么了；因为在安装那个模块时自会调用其 init_module()函数。可是，如果下层驱动程序是通过静态模块实现的话，那就要在这里处理下一层的初始化了。根据具体硬件的不同，sparcaudio 的下层有三种不同的驱动程序，另外为程序调试的目的还可以再加上一层虚设的驱动程序，所以这里有 4 个条件编译的语句。我们假设下层是通过可安装模块实现的，所以就跳过了这些静态模块的初始化。至于同一文件中，在撤销模块时调用的 sparcaudio_exit()，就留给读者自己阅读了。由于这个模块是静态连接的，代码中相应地还有一行：

==================== drivers/sbus/audio/audio.c 2231 2231 ====================

```
module_exit(sparcaudio_exit)
```

表示在系统退出运行时要拆除这个模块。

很自然地，我们在这里要看的第二个模块就是位于 sparcaudio 下层的模块，即“ amd7930”（我们假定声卡所用的芯片为 AMD7930）。这一次，我们假定 amd7930 是个动态安装模块，所以由 sys_init_module()在安装这个模块时调用它的 init_module()函数，其代码在 drivers/sbus/audio/amd7930.c

中：

==================== drivers/sbus/audio/amd7930.c 1677 1711 ====================

[sys_init_module()>init_module()]

```
/* Probe for the amd7930 chip and then attach the driver. */
#ifdef MODULE
int init_module(void)
#else
int __init amd7930_init(void)
#endif
{
	struct sbus_bus *sbus;
	struct sbus_dev *sdev;
	int node;

	/* Try to find the sun4c "audio" node first. */
	node = prom_getchild(prom_root_node);
	node = prom_searchsiblings(node, "audio");
	if (node && amd7930_attach(&drivers[0], node, NULL, NULL) == 0)
		num_drivers = 1;
	else
		num_drivers = 0;

	/* Probe each SBUS for amd7930 chips. */
	for_all_sbusdev(sdev, sbus) {
		if (!strcmp(sdev->prom_name, "audio")) {
			/* Don't go over the max number of drivers. */
			if (num_drivers >= MAX_DRIVERS)
				continue;

			if (amd7930_attach(&drivers[num_drivers],
					   sdev->prom_node, sdev->bus, sdev) == 0)
				num_drivers++;
		}
	}

	/* Only return success if we found some amd7930 chips. */
	return (num_drivers > 0) ? 0 : -EIO;
}
```

我们的目的不在于掌握 sbus 和 AMD7930 芯片本身的细节，所以就不详细深入到有关的代码中去了。从总体上说，这段程序所做的就是：对于探测到的每个 AMD7930 芯片，即音频信道，从结构数组 drivers[]中分配一个数据结构，然后调用 amd7930_attach()。如果一个信道也没有探测到，那就返回 -EIO 表示模块安装失败，否则就返回 0 表示安装成功。

函数 amd7930_attach()的代码在 drivers/sbus/audio/amd7930.c 中：

==================== drivers/sbus/audio/amd7930.c 1566 1660 ====================

[sys_init_module()>init_module()>amd7930_attach()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* Attach to an amd7930 chip given its PROM node. */
static int amd7930_attach(struct sparcaudio_driver *drv, int node,
                          struct sbus_bus *sbus, struct sbus_dev *sdev)
{
        struct linux_prom_registers regs;
        struct linux_prom_irqs irq;
        struct resource res, *resp;
        struct amd7930_info *info;
        int err;

        /* Allocate our private information structure. */
        drv->private = kmalloc(sizeof(struct amd7930_info), GFP_KERNEL);
        if (drv->private == NULL)
                return -ENOMEM;

        /* Point at the information structure and initialize it. */
        drv->ops = &amd7930_ops;
        info = (struct amd7930_info *)drv->private;
        memset(info, 0, sizeof(*info));
        info->ints_on = 1; /* force disable below */

        drv->dev = sdev;

        /* Map the registers into memory. */
        prom_getproperty(node, "reg", (char *)&regs, sizeof(regs));
        if (sbus && sdev) {
                resp = &sdev->resource[0];
        } else {
                resp = &res;
                res.start = regs.phys_addr;
                res.end = res.start + regs.reg_size - 1;
                res.flags = IORESOURCE_IO | (regs.which_io & 0xff);
        }
        info->regs_size = regs.reg_size;
        info->regs = sbus_ioremap(resp, 0, regs.reg_size, "amd7930");
        if (!info->regs) {
                printk(KERN_ERR "amd7930: could not remap registers\n");
                kfree(drv->private);
                return -EIO;
        }

        /* Put amd7930 in idle mode (interrupts disabled) */
        amd7930_idle(info);

        /* Enable extended FIFO operation on D-channel */
        sbus_writeb(AMR_DLC_EFCR, info->regs + CR);
        sbus_writeb(AMR_DLC_EFCR_EXTEND_FIFO, info->regs + DR);
        sbus_writeb(AMR_DLC_DMR4, info->regs + CR);
        sbus_writeb(/* AMR_DLC_DMR4_RCV_30 | */ AMR_DLC_DMR4_XMT_14,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1615                info->regs + DR);
1616
1617        /* Attach the interrupt handler to the audio interrupt. */
1618        prom_getproperty(node, "intr", (char *)&irq, sizeof(irq));
1619        info->irq = irq.pri;
1620        request_irq(info->irq, amd7930_interrupt,
1621                SA_INTERRUPT, "amd7930", drv);
1622        enable_irq(info->irq);
1623        amd7930_enable_ints(info);
1624
1625        /* Initalize the local copy of the MAP registers. */
1626        memset(&info->map, 0, sizeof(info->map));
1627        info->map.mmr1 = AM_MAP_MMR1_GX | AM_MAP_MMR1_GER |
1628                  AM_MAP_MMR1_GR | AM_MAP_MMR1_STG;
1629          /* Start out with speaker, microphone */
1630          info->map.mmr2 |= (AM_MAP_MMR2_LS | AM_MAP_MMR2_AINB);
1631
1632        /* Set the default audio parameters. */
1633        info->rgain = 128;
1634        info->pgain = 200;
1635        info->mgain = 0;
1636        info->format_type = AUDIO_ENCODING_ULAW;
1637        info->Bb.input_format = AUDIO_ENCODING_ULAW;
1638        info->Bb.output_format = AUDIO_ENCODING_ULAW;
1639        info->Bc.input_format = AUDIO_ENCODING_ULAW;
1640        info->Bc.output_format = AUDIO_ENCODING_ULAW;
1641        amd7930_update_map(drv);
1642
1643        /* Register the amd7930 with the midlevel audio driver. */
1644        err = register_sparcaudio_driver(drv, 1);
1645        if (err < 0) {
1646             printk(KERN_ERR "amd7930: unable to register\n");
1647             disable_irq(info->irq);
1648             free_irq(info->irq, drv);
1649             sbus_iounmap(info->regs, info->regs_size);
1650             kfree(drv->private);
1651             return -EIO;
1652        }
1653
1654        /* Announce the hardware to the user. */
1655        printk(KERN_INFO "amd7930 at %lx irq %d\n",
1656              info->regs, info->irq);
1657
1658        /* Success! */
1659        return 0;
1660 }
```

同样，我们的目的不在于这段程序的细节，因为那完全取决于具体的硬件和具体驱动程序的设计。我们在这里关心的主要有两件事。其一是对函数 register_sparcaudio_driver()的调用（1644 行），这个函

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

数是由上层模块 sparcaudio 提供的，模块 amd7930 通过它向上层登记一个 sparcaudio_driver 结构指针，也就是（代表着）一个 AMD7930 芯片。其二（1582 行），是使这个数据结构中的一个指针 ops，指向在采用 AMD7930 芯片条件下 sparcaudio 设备各种操作的函数跳转结构，即 sparcaudio_operations 结构 amd7930_ops：

==================== drivers/sbus/audio/amd7930.c 1503 1564 ====================

```
1503 /*
1504  *   Device detection and initialization.
1505  */
1506
1507 static struct sparcaudio_operations amd7930_ops = {
1508      amd7930_open,
1509      amd7930_release,
1510      amd7930_ioctl,
1511      amd7930_start_output,
1512      amd7930_stop_output,
1513      amd7930_start_input,
1514      amd7930_stop_input,
1515      amd7930_sunaudio_getdev,
1516      amd7930_set_output_volume,
1517      amd7930_get_output_volume,
1518      amd7930_set_input_volume,
1519      amd7930_get_input_volume,
1520      amd7930_set_monitor_volume,
1521      amd7930_get_monitor_volume,
1522      NULL,              /* amd7930_set_output_balance */
1523      amd7930_get_output_balance,
1524      NULL,              /* amd7930_set_input_balance */
1525      amd7930_get_input_balance,
1526      amd7930_set_output_channels,
1527      amd7930_get_output_channels,
1528      amd7930_set_input_channels,
1529      amd7930_get_input_channels,
1530      amd7930_set_output_precision,
1531      amd7930_get_output_precision,
1532      amd7930_set_input_precision,
1533      amd7930_get_input_precision,
1534      amd7930_set_output_port,
1535      amd7930_get_output_port,
1536      NULL,              /* amd7930_set_input_port */
1537      amd7930_get_input_port,
1538      amd7930_set_encoding,
1539      amd7930_get_encoding,
1540      amd7930_set_encoding,
1541      amd7930_get_encoding,
1542      amd7930_set_output_rate,
1543      amd7930_get_output_rate,
1544      amd7930_set_input_rate,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    amd7930_get_input_rate,
    amd7930_sunaudio_getdev_sunos,
    amd7930_get_output_ports,
    amd7930_get_input_ports,
    NULL,                   /* amd7930_set_output_muted */
    amd7930_get_output_muted,
      NULL,                  /* amd7930_set_output_pause */
      NULL,                  /* amd7930_get_output_pause */
      NULL,                  /* amd7930_set_input_pause */
      NULL,                  /* amd7930_get_input_pause */
      NULL,                  /* amd7930_set_output_samples */
      NULL,                  /* amd7930_get_output_samples */
      NULL,                  /* amd7930_set_input_samples */
      NULL,                  /* amd7930_get_input_samples */
      NULL,                  /* amd7930_set_output_error */
      NULL,                  /* amd7930_get_output_error */
      NULL,                  /* amd7930_set_input_error */
      NULL,                  /* amd7930_get_input_error */
      amd7930_get_formats,
};
```

我们在这里还是列出了 amd7930_attach()的全部代码以及 amd7930_ops 的全部定义，使读者对物理层的驱动程序有个感性的认识。即使不深入到具体的函数中去，我们也能从 amd7930_attach()的代码中约略地看到：它将芯片中的寄存器映射到内存空间，对硬件进行初始化，为来自 AMD7930 芯片的中断请求设置好中断服务程序（amd7930_interrupt()），并打开中断，然后再向其上层登记代表给定芯片的数据结构。同时，从数据结构 amd7930_ops 中也约略可以看出对 AMD7930 芯片的操作有：打开/关断输入或输出、控制输入或输出的音量、控制监听音量、设置输入或输出信道的各种参数和对电平的分辨率、设置数字化及还原时采用的编码制式以及采样的频率等等。应用程序通常通过系统调用 ioctol()来启动这些函数。

函数 register_sparcaudio_driver()是由上层模块 sparcaudio 移出，供下层模块（如 amd7930）用来向它登记的。除此以外，有些需要在上层模块中为下层模块完成的操作，如某些资源的分配，也可以放在这个函数中进行（当然，也可以由下层模块自己完成）。所以，这个函数所做的事情并不只是登记下层模块，还包括为具体音频信道（即 amd7930 芯片）设置缓冲区等，所以代码比较长，我们不在这里列出其代码了。所谓登记，在这里主要是将一个 sparcaudio_driver 结构指针传递给 sparcaudio，使它可以通过这个数据结构访问具体 amd7930 信道的一些数据，并且可以通过结构中的指针 ops 找到对 amd7930 操作的各种函数指针。在 sparcaudio 模块中有个 sparcaudio_driver 结构指针数组，用来保存已向它登记的指针，从而维持与下层模块的联系，此数组的定义在 drivers/sbus/audio/audio.c 中给出：

==================== drivers/sbus/audio/audio.c 73 73 ====================

```
static struct sparcaudio_driver *drivers[SPARCAUDIO_MAX_DEVICES];
```

相应地，在 amd7930 模块中则有一个 sparcaudio_driver 结构数组，也叫 driver[]，定义于 drivers/sbus/audio/amd7930.c 中：

==================== drivers/sbus/audio/amd7930.c 118 118 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static struct sparcaudio_driver drivers[MAX_DRIVERS];
```

这两个数组虽然同名，但是分属两个不同的模块。两个模块都是 static，所以即使把两个模块都作为静态模块编译并且连接，也不会造成冲突。这里要指出，sparcaudio 中虽然有个 sparcaudio_driver 结构指针数组，这并不意味着系统中要安装许多个 amd7930 的模块，而是意味着系统中可以有许多个 amd7930 芯片。所以，amd7930 模块只有一个，但是它操作的对象，或者说操作的上下文，却可以有多个，而数组中的指针则指向由同一模块驱动的各个对象。

除登记 sparcaudio_driver 结构指针以外，register_sparcaudio_driver()中还有件重要的事情，那就是为每个 amd7930 芯片在文件系统中创建文件节点，使应用程序可以“看见”这个具体的设备。就 devfs 设备文件系统而言，这是通过 devfs_register()完成的，在由 sparcaudio 模块所创建的目录“sound”下为每个 amd7930 芯片创建文件节点，如“sound/audio1”、“sound/audio2”等等。实际上，amd7930 是个很复杂的芯片，根据其结构可以进一步划分成混音、数字信号处理、音频控制 / 状态等功能模块；所以在 sparcaudio 模块的设计中选择将次设备号编码表示不同的功能模块，而为每一个功能模块都创建一个文件节点，如 sound/audio1、sound/mixer1 以及 sound/dsp1 等等。当然，也可以选择将它们合在一起。于是，每个这样的文件节点都对应着一个次设备号，而每个路径名则对应着主设备号与次设备号的一种组合。

总之，每个设备驱动模块在安装后都要使它的上层模块能“看见”它，有时还要使应用程序也能“看见”它。前者通过向上层模块登记而实现，后者则通过 mknod()、devfs_mk_dir()、devfs_register()一类的函数实现；并且既可以由下层模块自己完成，也可以由其上层模块替它完成，就如这里 sparcaudio 替 amd7930 创建文件节点那样。实际上，后者在本质上也是一种登记的过程，例如 mknod()就可以看成是由内核提供、让各种模块向文件系统登记的函数。通常一个模块堆栈中至少有一个模块需要为应用程序所见。

作为实例，我们再看一下 amd7930 模块的 cleanup module()：

==================== drivers/sbus/audio/amd7930.c 1713 1723 ====================

```
#ifdef MODULE
void cleanup_module(void)
{
    register int i;

    for (i = 0; i < num_drivers; i++) {
        amd7930_detach(&drivers[i]);
        num_drivers--;
    }
}
#endif
```

显然，amd7930_detach()是 amd7930_attach()的逆操作：

==================== drivers/sbus/audio/amd7930.c 1662 1675 ====================

[cleanup_module()>amd7930_detach()]

```
#ifdef MODULE
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/* Detach from an amd7930 chip given the device structure. */
static void amd7930_detach(struct sparcaudio_driver *drv)
{
    struct amd7930_info *info = (struct amd7930_info *)drv->private;

    unregister_sparcaudio_driver(drv, 1);
    amd7930_idle(info);
    disable_irq(info->irq);
    free_irq(info->irq, drv);
    sbus_iounmap(info->regs, info->regs_size);
    kfree(drv->private);
}
#endif
```

这里 unregister_sparcaudio_driver()是由 sparcaudio 模块移出、供下层模块撤销登记用的。可想而知，它从 devfs 文件系统中删去相应的文件节点，释放有关的缓冲区，并从 sparcaudio 的指针数组 drivers[]中将相应的表项恢复成 NULL。撤销向上层的登记以后，还要将有关硬件设置成“空转”状态，并与中断向量脱钩、以及撤销硬件寄存器的内存映象。总之，要消除 amd7930_attach()的所有影响，使系统恢复到相应的 amd7930_attach()操作之前的状态。

可安装模块的作用并不局限于设备驱动程序的实现，对作为可选项的许多文件系统（格式）的支持就常常是通过可安装模块实现的。此外，socket 机制也可以选择通过可安装模块来实现。至于各种网络规程的实现，那就更是非可安装模块莫属了。当然，网络规程的实现也可以看作是广义的设备驱动程序。另一方面，模块在应用程序界面上的表现（即文件节点）也并不一定在/dev 目录下或 devfs 子树中，如代表着插口的文件节点就通常不在这些目录中。

一个模块也并不限于只由一个文件节点代表，许多模块通过 create_proc_read_entry()在/proc 子树中也创建一个只读的文件节点，供用户或应用程序读取有关的状态和统计信息。更进一步，可安装模块甚至可以根本不与/dev 目录和 devfs 打交道，也不使用主设备号和次设备号，甚至不向上层模块登记，而只是通过 proc_register()在/proc 文件系统中创建一个文件节点，内核通过/proc 文件系统就可以访问到具体的模块。实际上，proc_register()是将向上层模块（/proc 文件系统）和向应用程序界面的登记（/proc 子树中的文件节点）合二为一了，所以只适合用于高层模块或一共只有一层的模块，总之是直接与 vfs 层接口的模块。这个函数既可以在/proc 文件系统中创建文件节点，也可以创建目录节点和符号连接节点，因此可以用来在/proc 目录下建立起类似于 devfs 的多层结构。由于在这种文件节点中并不使用主设备号和次设备号，就特别适用于一些高层的，严格说来算不上设备驱动的模块（如高层的网络规程），以及一些不适合（或有困难）为之分配主 / 次设备号的模块。缺点是它只支持 read()、write()和 lseek()三种文件操作，函数 proc_register()的代码在 fs/proc/generic.c 中：

==================== fs/proc/generic.c 350 375 ====================

```
static int proc_register(struct proc_dir_entry * dir, struct proc_dir_entry * dp)
{
    int  i;

    i = make_inode_number();
    if (i < 0)
```

```
                return -EAGAIN;
        dp->low_ino = i;
        dp->next = dir->subdir;
        dp->parent = dir;
        dir->subdir = dp;
        if (S_ISDIR(dp->mode)) {
                if (dp->proc_iops == NULL) {
                        dp->proc_fops = &proc_dir_operations;
                        dp->proc_iops = &proc_dir_inode_operations;
                }
                dir->nlink++;
        } else if (S_ISLNK(dp->mode)) {
                if (dp->proc_iops == NULL)
                        dp->proc_iops = &proc_link_inode_operations;
        } else if (S_ISREG(dp->mode)) {
                if (dp->proc_fops == NULL)
                        dp->proc_fops = &proc_file_operations;
        }
        return 0;
}
```

参数 dir 指向父目录的 proc_dir_entry 数据结构，如果父目录就是/proc 则为&proc_root。另一个参数 dp，则指向待创建节点的 proc_dir_entry 结构。在/proc 文件系统中不管是目录节点还是文件节点都使用同一种数据结构，即 proc_dir_entry，并且该结构中包含了通常分布在 dentry 和 inode 两种数据结构中的信息。读者可回顾一下“ /proc 特殊文件系统”一节中的有关内容。在调用 proc_register()之前，可安装模块要分配并初步设置好一个 proc_dir_entry 数据结构。这个数据结构中有三个函数指针 read_proc、write_proc 以及 get_info，应分别指向模块中的相应函数，它们的界面定义于 include/linux/proc_fs.h：

==================== include/linux/proc_fs.h 47 51 ====================

```
typedef   int (read_proc_t)(char *page, char **start, off_t off,
                          int count, int *eof, void *data);
typedef   int (write_proc_t)(struct file *file, const char *buffer,
                           unsigned long count, void *data);
typedef int (get_info_t)(char *, char **, off_t, int);
```

其中 get_info 也是用于读文件的，与 read_proc 的不同之处仅在于函数的调用界面。相比之下，read_proc 多了最后两个参数，一个是指针 eof，用来返回表示文件是否已经读到了结尾，另一个指针 data 就是 proc_dir_entry 结构中的指针 data，可以用来传递一些与特定数据结构有关的信息，例如类似于次设备号那样的数据结构序号。不过，这两个函数中只能选择其一，用了 read_proc 就不用 get_info。

在函数 proc_register()中，根据待创建节点的性质（模式）而设置其 proc_dir_entry 结构中的若干指针。如果是目录节点就设置其 proc_fops 和 proc_iop 两个指针，使它们分别指向/proc 文件系统的 inode_operations 数据结构和用于目录节点的 file_operations 数据结构；如果是符号连接节点就使 proc_iops 指向/proc 用于符号连接的 inode_operations 数据结构；而如果是文件节点则使 proc_fops 指向/proc 用于文件节点的 file_operations 数据结构。这几个数据结构都是/proc 文件系统所固有的。前三个

PDF-XCHANGE Click to buy NOW! www.docu-track.com

数据结构中的函数指针保证了当应用程序通过系统调用 open()在内核中启动 path_walk()时能找到/proc 文件系统中的节点，而最后一个数据结构中的函数指针，则提供了进入具体模块进行读 / 写的“中转站”。当打开一个/proc 文件时，内核会在 proc_lookup()中通过 proc_get_inode()为目标节点创建 inode 数据结构，把指向这个数据结构的指针 proc_fops 复制到 inode 结构中（指针 i_fop），然后又复制到 file 结构中（指针 f_op）。

我们来看看这个数据结构，这是在 fs/proc/generic.c 中定义的：

==================== fs/proc/generic.c 36 40 ====================

```
static struct file_operations proc_file_operations = {
	llseek:		proc_file_lseek,
	read:		proc_file_read,
	write:		proc_file_write,
};
```

从这个结构中可以看出，这个机制仅支持 llseek、read 和 write 三种文件操作，而不支持 ioctl。在设备驱动程序中，ioctl 是个很灵活、容量很大，因而功能很强、很重要的操作。例如，在前面的 amd7930 模块中定义了那么多的低层操作，而其离开了对 ioctl()系统调用的支持就根本无法实现。因此，缺少对 ioctl 的支持是这个机制的一个缺点。当然，以后在更新的版本中也许会把这一点考虑进去（再说那也是很容易的事）。

其次，结构中的函数指针都指向由/proc 文件系统提供的通用性程序，而不是指向具体模块中的有关函数。以读操作为例，跳转时的第一站为 proc_file_read()，其代码在同一文件中：

==================== fs/proc/generic.c 49 121 ====================

[sys_read()>proc_file_read()]

```
static ssize_t
proc_file_read(struct file * file, char * buf, size_t nbytes, loff_t *ppos)
{
	struct inode * inode = file->f_dentry->d_inode;
	char 	*page;
	ssize_t	retval=0;
	int	eof=0;
	ssize_t	n, count;
	char	*start;
	struct proc_dir_entry * dp;

	dp = (struct proc_dir_entry *) inode->u.generic_ip;
	if (!(page = (char*) __get_free_page(GFP_KERNEL)))
		return -ENOMEM;

	while ((nbytes > 0) && !eof)
	{
		count = MIN(PROC_BLOCK_SIZE, nbytes);

		start = NULL;
		if (dp->get_info) {
			/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
             * Handle backwards compatibility with the old net
             * routines.
             */
            n = dp->get_info(page, &start, *ppos, count);
            if (n < count)
                eof = 1;
        } else if (dp->read_proc) {
            n = dp->read_proc(page, &start, *ppos,
                          count, &eof, dp->data);
        } else
            break;

        if (!start) {
            /*
             * For proc files that are less than 4k
             */
            start = page + *ppos;
            n -= *ppos;
            if (n <= 0)
                break;
            if (n > count)
                n = count;
        }
        if (n == 0)
            break;   /* End of file */
        if (n < 0) {
            if (retval == 0)
                retval = n;
            break;
        }

        /* This is a hack to allow mangling of file pos independent
         * of actual bytes read.  Simply place the data at page,
         * return the bytes, and set `start' to the desired offset
         * as an unsigned int. - Paul.Russell@rustcorp.com.au
         */
        n -= copy_to_user(buf, start < page ? page : start, n);
        if (n == 0) {
            if (retval == 0)
                retval = -EFAULT;
            break;
        }

        *ppos += start < page ? (long)start : n; /* Move down the file */
        nbytes -= n;
        buf += n;
        retval += n;
    }
    free_page((unsigned long) page);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
120         return retval;
121  }
```

在第 69～80 行中，根据函数指针 get_info 和 read_proc 是否为 NULL 而调用其中之一（或二者都不调用），这才进入了由模块所提供的函数中。经过这样一层中转，程序的执行效率多少受到一点影响。相比之下，如果模块直接向内核登记其 file_operations 数据结构（像 sparcaudio 那样），则在经由 file_operations 结构跳转时直接就进入了由模块提供的函数。

但是，尽管如此，将文件节点创建在/proc 文件系统中还是很有吸引力的。摆脱了对主 / 次设备号的依赖，就使可安装模块的设计变得更灵活，并因不再有设备号惟一性的问题而更可移植，程序设计也要简单一些。

那么，是否可以像对待普通文件那样，把代表可安装模块的文件节点创建在文件系统中任意（除/proc 文件以外）的位置上呢？如果不使用主 / 次设备号的话，至少在目前的内核上是不行的。这里的困难在于 file_operations 数据结构是每个文件系统（类型）一个，而不是每个文件一个（devfs 文件系统是个例外），再说可安装模块也并不是一种特殊的文件类型。当然，如果仍旧使用主设备号 / 次设备号，那是可以的。可正是因为使用设备号才更有必要把这些文件集中在一起，因为那样才便于观察到设备号的冲突。

8.4　PCI 总线

说到外部设备的驱动就离不开系统的外设总线。最初 PC 机中的外设总线只是 8 位的，就是说每次只能读 / 写一个字节；后来扩充成 16 位，称为 ISA 总线；再后来又扩充成 32 位的 EISA 总线。但是，随着技术的发展和应用的日益普及，人们逐渐认识到这些总线都存在着一些重要的、根本性的缺点，因而需要开发出一种全新的总线。当时提出了两种主要的候选总线结构，一种称为 VESA，另一种就是 PCI。经过一段时期的竞争，PCI 总线成为了事实上的标准总线，不光是 PC 系统结构中的标准总线，也是许多其它系统结构中的标准总线。现在的 PC 机一般都提供若干 PCI 总线插槽，同时也提供少量 ISA 总线插槽以求跟旧式接口卡的兼容。至于 EISA，则还来不及推广就被 PCI 取代了。以前，外设总线一般都是与具体的 CPU 和系统结构密切联系的，CPU 不同、系统结构不同，采用的总线也就不同。例如 ISA 总线就是用于采用 Intel X86 系列 CPU 的 PC 中，UNIBUS 和 Qbus 就是用于以前的 PDP-11 中，而 VME 总线则用于 Motorola 的 M68K 和 Power PC 系列的系统中。这一点可以说是历来如此。可是，自从 PCI 总线问世以后，却很快就成为了通用的标准总线，以至于不管是什么 CPU，不管是为哪种外设开发的芯片组，都会提供跟 PCI 总线的接口。从这个意义上说，似乎 PCI 总线成了计算机系统结构的中心，而 CPU 倒反而退居从属的地位了。随着多处理器 SMP 结构的采用和普及，这种趋向就更明显了，因为系统中可以有许多个处理器，而 PCI 总线却往往只有一条。然而，PCI 总线的标准，即其规格书，是个不太好读、不太好理解的文本。拿到一块 PCI 接口卡或者一组芯片，要从规格书或厂商提供的说明书出发，开发出该项设备的驱动程序实非易事。在这个意义上，Linux 内核中有关的源代码恰恰为我们深入理解和运用 PCI 总线（及设备）提供了一个实例、一个样板。

以后，为行文的方便，在不至于引进混淆的场合我们将直接以“总线”或“PCI”表示“PCI 总线”。

那么，ISA 总线（以及 PCI 以前的那些总线）到底有些什么缺点呢？换言之，PCI 总线有些什么优

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

点呢？

首先，ISA 乃至 EISA 的速度都太慢。这两种总线的时钟频率都是 8.33MHz，就算是 32 位的 EISA，其理论上的最大通量也不过是每秒 33MB（ISA 的最大通量只是 8.33MB)。这显然不能满足图像、网络等方面的应用需要。例如，光是一个 100 兆的 Ethernet 接口，其理论上的最大通量就已经超过 12MB 了。但是，提高总线的速度并不单纯是个提高时钟频率的问题。在高速的条件下，有不少物理问题（如电信号的传输）要考虑和解决。而且，如果采用相同的插槽则还要考虑跟已经存在的接口卡兼容的问题。所以，PCI 总线采用了完全不同的插槽，时钟频率则提高到 33MHz，使理论上的最大通量提高到 133MB。而且，还为进一步把时钟频率提高到 66MHz、总线宽度提高到 64 位留下了余地。

其次是地址的分配与设置。就像存储器本身是一种资源一样，存储器的地址也是一种资源。在同一时间内，一个地址只能惟一地用于一个物理的存储单元，或者就空闲不用。在 i386 系统结构中，对内存的访问和对输入 / 输出寄存器的访问通过两套不同的指令完成，所以有存储器和 I/O 两个不同的地址空间。一般而言，内存的物理地址以及输入 / 输出寄存器的地址是由硬件决定的，不过对于内存的物理地址还可以通过地址映射机制来一次转换。可是，怎样处理外部设备上的存储空间呢？ISA（以及其它早期的总线）接口卡上都采用一些跳线或小开关，将接口卡插上总线前都要先通过这些小开关设置好地址（还有中断请求号），而对于软件则也要在一个“安装”过程中加以设置，使二者相符。可是，这样做不但麻烦，还往往成为系统因为地址冲突或不符而不能正常运行的原因。再说，随着外部设备的日益复杂，这样做已经不大现实了。所以，理想的办法是由系统软件自动设置，总的思路是：

(1) 每块接口卡（每项外设）都通过某种途径告诉系统：卡上有几个存储区间及 I/O 地址区间，每个区间是多大，以及各自在卡上（本地）的地址。这些地址在本质上都是局部的、内部的，所以都从 0 起算。但是，这些区间不与总线直接相连，在把插口卡插上总线并加电之初，从总线上还访问不到这些区间，所以不会互相冲突。为区别这种地址，我们不妨称之为“卡上地址”，虽然实际上未必是在接口卡上，有些外设芯片其实是固定在母板上的。

(2) 系统软件在知道了一共有多少外部设备、各自又有什么样的存储区间以后，就可以统筹地为这些区间分配“物理地址”，并且建立起这些区间与总线之间的连接，以后就可以通过这些地址来访问。显然，这里所谓“物理地址”与真正的物理地址是有些区别的，它实际上也是一种逻辑地址，所以常称为“总线地址”，因为这是 CPU 在总线上所看到的地址。可想而知，接口卡上一定有着某种地址映射机制。所谓“为外设分配地址”，就是为其分配总线地址，并为之建立起映射。这种映射代替了从前的跳线或小开关，卡上有几个地址区间就有几个映射。对于 CPU，“总线地址”就相当于物理地址，还可以通过虚存地址的映射再加一次变换。此外，对于中断请求线的连接也与此相似。

事实上，PCI 总线正是这样设计的。此外，对于 I/O 地址空间与内存地址空间相分离的系统结构，如 i386 结构，还可以选择将 I/O 寄存器的地址也映射成内存地址（总线地址），这样就可以通过访内指令来操作这些寄存器了。这一方面有可能简化程序设计，另一方面也可避免 i386 系统结构中 I/O 地址空间太小（16 位）、太拥挤的问题。

可以想像，每个 PCI 设备或者接口卡上一定有许多用来完成这个过程，即建立起这些连接和映射的寄存器，系统或设备初始化的时候要通过这些寄存器来“配置”该设备的各个总线地址区间。可是，这些寄存器本身的地址又怎么办呢？这不是又回到了原先的问题吗？下面读者就会看到，PCI 总线的设计者对这个问题解决得很好（但是从程序设计的角度却有些令人头痛）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第三个问题是对使用总线的竞争。在早期的计算机系统中，整个系统只有一个 CPU，从总线的角度来说，这个 CPU 就是“主设备”（Master），而其它的都是“从设备”，只有主设备才能启动跨总线的操作。需要由“从设备”启动的操作也是有的，那就是对内存的 DMA 操作。此时“从设备”先向 CPU 发出一个 DMA 请求，让 CPU 暂停访问内存，实际上是暂停包括访问总线在内的一切外部操作，使系统中暂时没有了“主设备”。得到允许以后，这“从设备”就暂时升级变成了对内存的“主设备”，从而可以启动内存操作。这种内存操作当然是跨总线的，但是因为 CPU 已暂停活动，所以不存在竞争使用总线的问题。或者说，对（使用）总线的竞争寓于对（使用）内存的竞争之中。可是，在多处理器的系统中，对总线的竞争就成为问题了，如果两个 CPU 同时启动跨总线的访问，怎样来解决冲突呢？还有，随着技术的发展，一些“从设备”，即外设接口卡也带上了智能，有了本地的处理器，在这样的情况下，应该允许一个“从设备”直接访问另一个“从设备”上的存储区间或 I/O 寄存器，而不必由 CPU 介入。这与 DMA，即“（由外设）直接访问内存”，是同样的概念。可是，如果一个“从设备”要直接访问另一个“从设备”，那当然也要先取得对总线的使用权，暂时变成总线的主设备。为了解决竞争使用总线的问题，PCI 总线上配备了一个仲裁器。遇有冲突时，仲裁器会选择其中之一（包括 CPU）暂时成为当前的“主设备”，而其它的则只好等待。由此又生出另一个问题来，如果 CPU，或一个设备，想要启动的是写操作，但是由于冲突而一时不能成为总线的“主设备”，那么它是只能停下来等待；还是可以让它把要写的内容放在一个缓冲区中，把它托付给 PCI 总线，自己则接着继续运行？显然这对于 CPU 的效率是有影响的。PCI 总线的设计考虑到了这个问题，为写操作提供了缓冲。当然，对于读操作就没有办法，只好等待了。所以，从效果上看，跨 PCI 总线的写操作往往比读操作快。可想而知，在这个方面，PCI 总线的硬件（芯片）设计是相当复杂的，但好在对软件是“透明”的，所以在代码中看不到这个问题的存在。但是，允许“从设备”通过竞争暂时变成“主设备”，则实际上是对 DMA 的推广，因而对外部设备接口的设计与实现，进而对设备驱动程序的代码有着重要的影响。我们将在“块设备驱动”一节中结合 DMA 操作进一步讨论这个问题。

还有个问题是总线的扩充问题。一块母板上能容纳的插槽、或不经插槽直接与 PCI 总线相连的芯片的数量总是有限的，能不能在需要时通过一块接口卡对总线的容量加以扩充，或者连接上另一条（同种或异种）总线呢？读者也许觉得这个问题很简单，其实不然（不过我们在这里就不深入讨论了）。PCI 总线在这方面也解决得很好。人们设计出了各种各样的“ PCI 桥”芯片，通过一个 PCI 桥就可以连接到一条 PCI 总线。CPU 通过“宿主—PCI 桥”与一条 PCI 总线相连，处在这种位置上的 PCI 总线称为“主（Primary）PCI 总线”，或者就称“主总线”。PC 机中通常只有一个“宿主—PCI 桥”，但是在特殊的系统结构中也可以有多个。在一条 PCI 总线的基础上，可以再通过“ PCI 桥”连接到其它次一层的总线，例如通过“ PCI-PCI 桥”可以连接到另一条 PCI 总线，通过“ PCI-ISA 桥”可以连接到一条 ISA 总线。事实上，现代 PC 机中的 ISA 总线正是通过“ PCI-ISA 桥”连接在 PCI 总线上的。这样，通过使用“ PCI-PCI 桥”，就可以构筑起一个层次的、树状的 PCI 系统结构。对于上层的总线而言，连接在这条总线上的 PCI 桥也是一个设备。但是，这是一种特殊的设备，它既是上层总线上的一个设备，实际上又是上层总线的延伸。

实际上，在 PC 机中，“宿主—PCI 桥”与内存控制器往往都做在同一个芯片中。CPU 与这个器件、以及通过这个器件与内存的连接就是原来意义上的“系统总线”，而通过这个器件直接相连的就是系统的主 PCI 总线。所有的外设，包括磁盘、键盘、鼠标器、并行口、串行口、网络卡、等等，全都直接或间接地接在 PCI 总线上。其中有些以接口卡和插槽的形式相连，有的则固定在系统母板上通过电子

PDF-XCHANGE Click to buy NOW! www.docu-track.com

线路直接相连。特别地，PC 机中一般还通过一个“PCI-ISA 桥”连接到一条 ISA 总线，提供若干 ISA 插槽。不过，插在 ISA 总线上的接口卡就不具备 PCI 的地址映射功能了。此外，如前所述，需要时还可以通过“PCI-PCI 桥”连接到其它 PCI 总线。这样，如果把 CPU 比喻作一个城市的中心，系统总线就好像是“一环路”，主 PCI 总线就好像是“二环路”，而 ISA 总线以及连在主 PCI 总线上的其它 PCI 总线就是“三环路”了。图 8.2 所示是个 PC 系统结构的示意图。

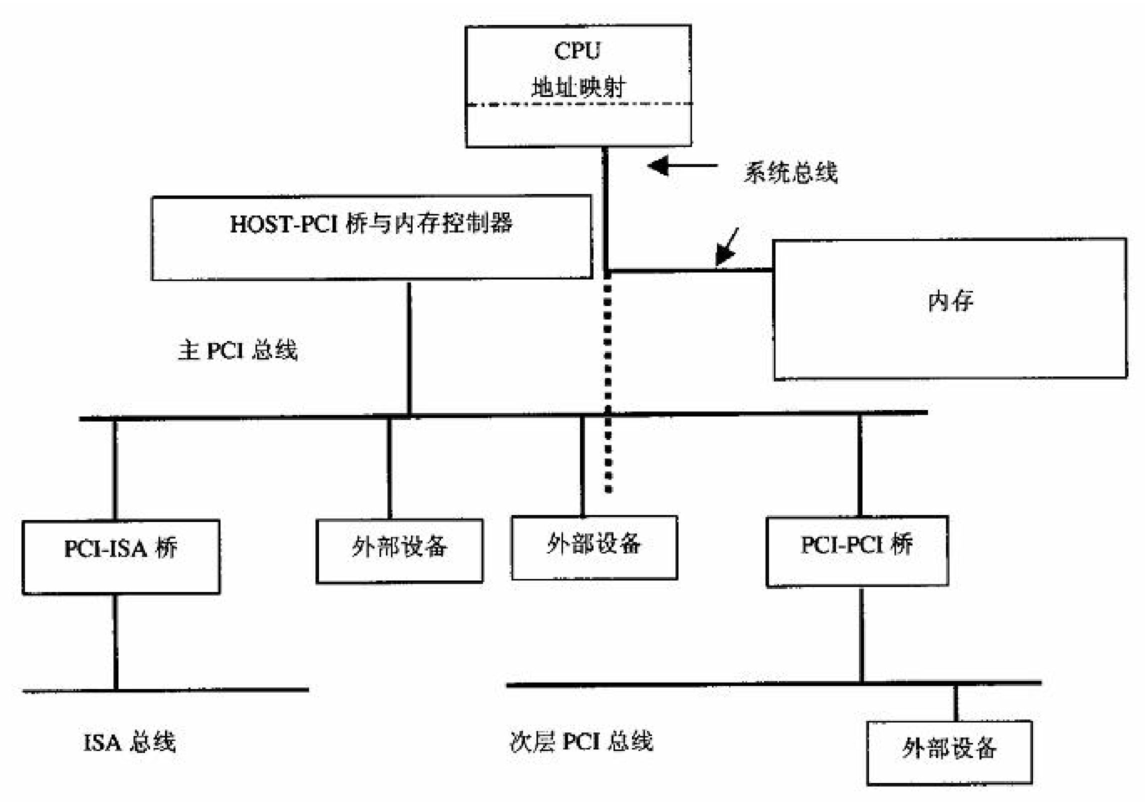


图 8.2　PC 系统结构与 PCI 总线

图 8.2 中在系统总线与 PCI 总线上的一个设备之间有条虚线，表示 PCI 总线上的设备在完成了配置以后就好像连接在系统总线上一样。而在此之前则对于系统总线和 PCI 总线都是不可见的，所以不会互相冲突。对于连接在次层 PCI 总线上的设备也是一样。此外，系统的内存控制器与 HOST-PCI 桥通常都集成在同一芯片中，可以把内存也看作是连在 PCI 总线上的一个设备，不过它永远是“从设备”。

从上面的介绍可以看出，PCI 总线的设计考虑到了种种方面的问题，并且解决得很好，它之所以能成为受到普遍接受的标准总线绝不是偶然的。

前面讲过，每个 PCI 设备或者接口卡上有许多用于地址配置的寄存器，初始化时要通过这些寄存器来“配置”该设备的总线地址。一旦完成了配置以后，CPU 就可以访问该设备的各项资源，就好像那是内存的一部分一样，这就没有什么特殊的了。所以，本节的重点是配置 PCI 设备的过程，与这个过程有关的代码并不简单。在本书中，我们把每个设备上的这些寄存器合在一起称为该设备的“配置寄存器组”。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PCI 标准规定每个设备的配置寄存器组最多可以有 256 字节的连续空间，其中开头 64 个字节的用途和格式是标准的，称为配置寄存器组的“头部”（configuration header）。这样的“头部”又有两种，其中“0 型”（type 0）头部用于一般的 PCI 设备，“1 型”头部则用于各种 PCI 桥。但是，不管是“0 型”还是“1 型”，其开头 16 个字节的用途和格式总是共同的。这 16 个字节中包含着有关头部的类型、设备的种类、设备的一些性质、由谁制造等等信息。对于这 16 个字节的地址，include/linux/pci.h 中定义了这样一些常数：

==================== include/linux/pci.h 24 26 ====================

```
#define PCI_VENDOR_ID       0x00 /* 16 bits */
#define PCI_DEVICE_ID       0x02 /* 16 bits */
#define PCI_COMMAND         0x04 /* 16 bits */
    . . . . . .
```

==================== include/linux/pci.h 38 38 ====================

```
#define PCI_STATUS     0x06 /* 16 bits */
    . . . . . .
```

==================== include/linux/pci.h 54 62 ====================

```
#define PCI_CLASS_REVISION  0x08 /* High 24 bits are class, low 8
                           revision */
#define PCI_REVISION_ID       0x08   /* Revision ID */
#define PCI_CLASS_PROG        0x09   /* Reg. Level Programming Interface */
#define PCI_CLASS_DEVICE      0x0a   /* Device class */

#define PCI_CACHE_LINE_SIZE 0x0c /* 8 bits */
#define PCI_LATENCY_TIMER   0x0d /* 8 bits */
#define PCI_HEADER_TYPE     0x0e /* 8 bits */
```

这里的寄存器 PCI_HEADER_TYPE 表明是哪一种头部，PCI_CLASS_DEVICE 和 PCI_CLASS_PROG 则表明是哪一种设备。例如，PCI_CLASS_DEVICE 的高 8 位为 0x02 表示是网络设备，而低 8 位为 0 且 PCI_CLASS_PROG 为 0，又进一步表示这是 Ethernet 接口卡。又如，PCI_CLASS_DEVICE 的高 8 位为 0x07 表示是“简单通信控制器”，低 8 位为 01 进一步表示是并行口，而 PCI_CLASS_PROG 为 0 表示单向、为 1 表示双向、为 2 表示符合 ECP 1.0 规定（IEEE1284 并行口标准中的一种操作模式）。还有，PCI_VENDOR_ID 表示由哪一家厂商制造，如 Intel 的 ID 号码是 0x8086、Compaq 的 ID 号码是 0x0e11。这些号码是由一个统一的机构分配、指定的，所以不会重复。每家厂商对其产品也有编号，那就是寄存器 PCI_DEVICE_ID 中的内容。所有这些信息都是由厂商固化在产品中的，不会变化。在 Linux 系统上，可以通过“cat /proc/pci”查看系统中所有 PCI 设备的类别、型号以及厂商等等信息，那就是从这些寄存器来的。

所以，根据这些信息就可以确定应该怎样进一步解释和处理其余的 48 个字节。至于头部以外的 192 个字节，则取决于具体的设备，如果不需要也可以没有。除地址配置以外，这些（头部）寄存器还有个重要的作用，就是使 CPU 能够探测到相应设备（接口）的存在，并且确定该设备的种类和一些特性，包括由谁制造等等。这样，用户就不再需要通过种种途径告知系统都有哪一些外设，而改由 CPU 通过一个称为“枚举”的过程自动扫描探测所有连接在 PCI 总线上的外设。

如前所述，这些寄存器本身的地址就是一个问题。首先，很难为不同的设备指定不同的起始地址，因为那样就又回到了原来的问题上。试想，每个设备都可以有 256 字节的配置寄存器组，如果要为每

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

一种可能的设备都保留不同的地址，那么 1024 种设备就得保留 256KB，一万种呢？更多呢？何况这又给地址的管理带来了麻烦，这显然是不现实的。比较好的办法是让所有设备的配置寄存器组都采用相同的地址，由所在总线的 PCI 桥在访问时附加上其它条件来区分。而 CPU 则通过一个统一的入口地址向“宿主—PCI 桥”发出命令，由相应的 PCI 桥间接地完成具体的读写。对于 i386 结构的处理器，PCI 总线的设计者在 I/O 地址空间保留了 8 个字节用于这个目的，那就是 0xCF8～0xCFF。这 8 个字节实际上构成两个 32 位的寄存器，第一个是“地址寄存器 ” 0xCF8，第二个是“数据寄存器 ” 0xCFC。要访问某个设备中的某个配置寄存器时，CPU 先往地址寄存器中写入目标地址，然后通过数据寄存器读写数据。不过，写入地址寄存器中的目标地址是一种包括总线号、设备号、功能号以及配置寄存器地址在内的综合地址，其构成如图 8.3 所示。

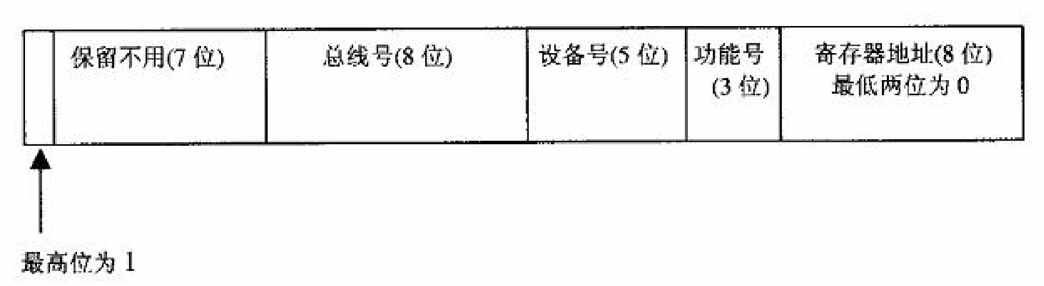


图 8.3　写入地址寄存器 0xCF8 的综合地址

这里的总线号、设备号以及功能号是对配置寄存器地址的扩充，就是上面说的附加条件。首先，每条 PCI 总线都有个总线号，主总线的总线号固定为 0，其余的则由 CPU 在枚举阶段每当探测到一个 PCI 桥时便为其指定一个，依次递增。设备号一般代表着一块 PCI 接口卡（更确切地说是 PCI 总线接口芯片），通常取决于插槽的位置。每块 PCI 接口卡上可以有若干个功能模块，这些功能模块共用同一个 PCI 总线接口芯片，包括其中用于地址映射的电子线路，以降低成本。从逻辑的角度说，每个“功能”实际上就是一项设备，所以设备号和功能号合在一起又可以称作“逻辑设备号”，而每块卡上最多可以容纳 8 个逻辑设备。显然，这些字段结合在一起就惟一地确定了系统中的一项 PCI 逻辑设备。读者也许会问：一开始的时候，在尚未为各条总线指定总线号之前，又怎样来访问特定的总线呢？事实上，一开始的时候只有 0 号总线是可访问的，在扫描 0 号总线时如果发现上面某一个设备是 PCI 桥，就为之指定一个新的总线号，例如 1，这样 1 号总线也就可以访问了，这就是枚举阶段的任务之一。

一般，每个逻辑设备中都有几个需要映射的地址区间，所以在 0 型头部中定义了 6 个“基地址 ”寄存器，可以分别用于 6 个地址区间的映射。下面读者就会看到，对这些寄存器的操作是相当复杂的。例如，直接读时是区间的地址加上一些标志位，而若先往里面写上全 1 再读就成了区间长度。除了这 6 个常规的区间外，逻辑设备中有时候还会有一块 ROM，所以又有一个“扩充 ROM 基地址”寄存器。此外，还有用来设置中断请求线的寄存器以及其它几个寄存器。

对于 0 型头部中这些寄存器的地址，include/linux/pci.h 中定义了一些常数：

==================== include/linux/pci.h 72 83 ====================

```
/*
 * Base addresses specify locations in memory or I/O space.
 * Decoded size can be determined by writing a value of
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * 0xffffffff to the register, and reading it back.  Only
 * 1 bits are decoded.
 */
#define PCI_BASE_ADDRESS_0  0x10 /* 32 bits */
#define PCI_BASE_ADDRESS_1  0x14 /* 32 bits [htype 0,1 only] */
#define PCI_BASE_ADDRESS_2  0x18 /* 32 bits [htype 0 only] */
#define PCI_BASE_ADDRESS_3  0x1c /* 32 bits */
#define PCI_BASE_ADDRESS_4  0x20 /* 32 bits */
#define PCI_BASE_ADDRESS_5  0x24 /* 32 bits */
```

==================== include/linux/pci.h 96 100 ====================

```
/* Header type 0 (normal devices) */
#define PCI_CARDBUS_CIS     0x28
#define PCI_SUBSYSTEM_VENDOR_ID  0x2c
#define PCI_SUBSYSTEM_ID    0x2e
#define PCI_ROM_ADDRESS     0x30 /* Bits 31..11 are address, 10..1 reserved */
    . . . . . .
```

==================== include/linux/pci.h 104 110 ====================

```
#define PCI_CAPABILITY_LIST 0x34 /* Offset of first capability list entry */

/* 0x35-0x3b are reserved */
#define PCI_INTERRUPT_LINE  0x3c /* 8 bits */
#define PCI_INTERRUPT_PIN   0x3d /* 8 bits */
#define PCI_MIN_GNT         0x3e /* 8 bits */
#define PCI_MAX_LAT         0x3f /* 8 bits */
```

以后，随着代码的阅读，读者自会明白这些寄存器的作用和有关的操作。

PCI 桥的作用和功能不同于一般设备，1 型头部中的寄存器自然也就不同。对于 1 型头部中的寄存器地址，include/linux/pci.h 中另有一些常数定义：

==================== include/linux/pci.h 112 118 ====================

```
/* Header type 1 (PCI-to-PCI bridges) */
#define PCI_PRIMARY_BUS     0x18 /* Primary bus number */
#define PCI_SECONDARY_BUS   0x19 /* Secondary bus number */
#define PCI_SUBORDINATE_BUS 0x1a /* Highest bus number behind the bridge */
#define PCI_SEC_LATENCY_TIMER    0x1b /* Latency timer for secondary interface */
#define PCI_IO_BASE         0x1c /* I/O range behind the bridge */
#define PCI_IO_LIMIT        0x1d
    . . . . . .
```

==================== include/linux/pci.h 123 125 ====================

```
#define PCI_SEC_STATUS      0x1e /* Secondary status register, only bit 14 used */
#define PCI_MEMORY_BASE     0x20 /* Memory range behind */
#define PCI_MEMORY_LIMIT    0x22
```

==================== include/linux/pci.h 128 129 ====================

```
#define PCI_PREF_MEMORY_BASE     0x24 /* Prefetchable memory range behind */
#define PCI_PREF_MEMORY_LIMIT    0x26
    . . . . . .
```

==================== include/linux/pci.h 134 142 ====================

```
#define PCI_PREF_BASE_UPPER32    0x28 /* Upper half of prefetchable memory range */
#define PCI_PREF_LIMIT_UPPER32   0x2c
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
136 #define PCI_IO_BASE_UPPER16 0x30 /* Upper half of I/O addresses */
137 #define PCI_IO_LIMIT_UPPER16     0x32
138 /* 0x34 same as for htype 0 */
139 /* 0x35-0x3b is reserved */
140 #define PCI_ROM_ADDRESS1    0x38 /* Same as PCI_ROM_ADDRESS, but for htype 1 */
141 /* 0x3c-0x3d are same as for htype 0 */
142 #define PCI_BRIDGE_CONTROL  0x3e
```

在 1 型头部中也有两个地址区间，所以前面定义的寄存器 PCI_BASE_ADDRESS_0 和 PCI_BASE_ADDRESS_1（见 78 和 79 行）也适用于 1 型头部。PCI 桥跨在两条总线之间，寄存器 PCI_PRIMARY_BUS 和 PCI_SECONDARY_BUS 的内容就说明了其上下两端的总线号，其中 PCI_SECONDARY_BUS 就是该 PCI 桥所连接和控制的总线，而 PCI_SUBORDINATE_BUS 则说明自此以下、在以此为根的子树中最大的总线号是什么。CPU 在枚举阶段所作的是深度优先的扫描，所以子树中的总线号总是连续递增的。当 CPU 往 I/O 寄存器 0xCF8 中写入一个综合地址以后，从 0 号总线开始，每个 PCI 桥会把综合地址中的总线号与其自身的总线号相比，如果相符就用逻辑设备号在本总线上寻访目标设备；否则就进一步把这个总线号与 PCI_SUBORDINATE_BUS 中的内容相比，如果目标总线号落在当前子树的范围中，就把综合地址传递给其下的各个次层 PCI 桥，要不然就不予理睬。这样，最终就会找到目标设备。如前所述，这个过程仅用于设备的配置阶段，一旦完成了配置，CPU 就直接通过有关的总线地址访问目标设备了。

同样，我们把 1 型头部中的其它一些寄存器也放到阅读有关代码时再介绍。

还有一种 2 型头部是用于“ PCI-CardBus 桥”的，CardBus 是笔记本电脑中使用的总线，我们在这里不感兴趣。

由于 PCI 总线的重要性和特殊性，PC 机制造商们在 BIOS 中提供了 PCI 总线操作，即 PCI 总线和设备的枚举以及配置所需的功能与服务。其中 PCI 总线和设备的枚举放在系统加电以后的自检阶段，而对配置过程所需的基本操作则以类似于 BIOS 调用的形式提供服务。考虑到现代的 PC 机操作系统都已采用保护模式和页式映射，BIOS 中还提供了一个特殊的 PCI 操作调用入口，供运行于保护模式和页式映射的 CPU 调用。提供了这些功能与服务的 BIOS 称为“ PCI BIOS”。早期的 Linux 内核也依靠 BIOS 完成对 PCI 的枚举和配置操作，但是后来已实现了自己的 PCI 总线操作而不再依赖于 BIOS。现在，是否采用 PCI BIOS 提供的服务是个条件编译选择项。我们在下面将阅读 Linux 内核自己的 PCI 总线操作的代码，因为只有这样才能真正搞清有关的机理，否则一进入 BIOS 就进了黑盒子，不知道到底在干些什么了。

不过，正因为 Linux 内核以前也采用 PCI BIOS 提供的服务，所以有很多函数的函数名都带有前缀 pcibios。尽管现在已经可以不再涉及 BIOS，却还是保留了其中一些函数原来的函数名不变，以免引起混乱。

我们先看几个低层的函数，这些函数就是通过寄存器 0xCF8 和 0xCFC 读写目标设备上的配置寄存器时使用的。对配置寄存器的读写可以是按字节、按 16 位字、按 32 位长字的读写。具体的函数由 gcc 在编译时的预处理中根据宏定义生成，这个宏定义在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 478 504 ====================

```
478 /*
479  * Wrappers for all PCI configuration access functions.  They just check
```

```
480  * alignment, do locking and call the low-level functions pointed to
481  * by pci_dev->ops.
482  */
483
484 #define PCI_byte_BAD 0
485 #define PCI_word_BAD (pos & 1)
486 #define PCI_dword_BAD (pos & 3)
487
488 #define PCI_OP(rw,size,type) \
489 int pci_##rw##_config_##size (struct pci_dev *dev, int pos, type value) \
490 {                                         \
491      int res;                             \
492      unsigned long flags;                           \
493      if (PCI_##size##_BAD) return PCIBIOS_BAD_REGISTER_NUMBER;    \
494      spin_lock_irqsave(&pci_lock, flags);                \
495      res = dev->bus->ops->rw##_##size(dev, pos, value);      \
496      spin_unlock_irqrestore(&pci_lock, flags);          \
497      return res;                                   \
498 }
499
500 PCI_OP(read, byte, u8 *)
501 PCI_OP(read, word, u16 *)
502 PCI_OP(read, dword, u32 *)
503 PCI_OP(write, byte, u8)
504 PCI_OP(write, word, u16)
```

以 501 行为例，经过 gcc 预处理的字符串替换就变成了函数 pci_read_config_word()的定义：

```
int pci_read_config_word(struct pci_dev *dev, int pos, u16* value)
{
    int res;
    unsigned long flags;
    if (PCI_word_BAD) return PCIBIOS_BAD_REGISTER_NUMBER;
    spin_lock_irqsave(&pci_lock, flags);
    res = dev->bus->ops->read_word(dev, pos, value);
    spin_unlock_irqrestore(&pci_lock, flags);
    return res;
}
```

同样地，其它几行就变成了 pci_read_config_byte()、pci_read_config_dword()、pci_write_config_byte()、pci_write_config_word()等函数的定义。

再看 pci_read_config_word()的代码。首先通过宏操作 PCI_word_BAD 检查地址 pos 是否与 16 位字边界对齐，然后就在加锁且不允许中断的条件下通过由具体设备提供的函数指针完成操作。这里参数 dev 指向代表着目标设备的 pci_dev 数据结构，读者在后面将会看到 pci_dev 数据结构的定义，这里我们先简单地介绍一下所涉及的几个字段。这个数据结构中有个指针 bus，指向代表着设备所在总线的 pci_bus 数据结构（见后），而 pci_bus 结构中又有个指针 ops，指向一个 pci_ops 数据结构，这个数据结构的定义在 include/linux/pci.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/pci.h 424 433 ====================

```
/* Low-level architecture-dependent routines */

struct pci_ops {
     int (*read_byte)(struct pci_dev *, int where, u8 *val);
     int (*read_word)(struct pci_dev *, int where, u16 *val);
     int (*read_dword)(struct pci_dev *, int where, u32 *val);
     int (*write_byte)(struct pci_dev *, int where, u8 val);
     int (*write_word)(struct pci_dev *, int where, u16 val);
     int (*write_dword)(struct pci_dev *, int where, u32 val);
};
```

显然，这是一个函数跳转表，与我们在前几章中看到过的一些数据结构相似。这个数据结构中提供的函数指针都是用来读 / 写 PCI 设备的配置寄存器的。内核中有三个 pci_ops 数据结构，分别用于所谓 " 1 型" 和 " 2 型" 的 PCI 配置寄存器操作（注意不是 1 型和 2 型的头部），以及通过 BIOS 完成的操作，均定义于 arch/i386/kernel/pci-pc.c。CPU 在初始化时根据条件编译或对硬件的测试从这三个数据结构中选择使用其一。在 PCI 总线发展的早期，有些 "宿主—PCI 桥" 曾经用过 " 2 型 " PCI 配置寄存器操作，但是后来在 PCI 总线标准中已经明文规定不再生产此类"宿主—PCI 桥"。所以，pci_direct_conf2 只是为了与可能还在使用中的某些老式母板兼容而设的，实际上一般都会选择 pci_direct_conf1。

==================== arch/i386/kernel/pci-pc.c 82 89 ====================

```
static struct pci_ops pci_direct_conf1 = {
     pci_conf1_read_config_byte,
     pci_conf1_read_config_word,
     pci_conf1_read_config_dword,
     pci_conf1_write_config_byte,
     pci_conf1_write_config_word,
     pci_conf1_write_config_dword
};
```

可见，上述的 pci_read_config_word() 是通过 pci_conf1_read_config_word() 完成的，其代码在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 45 50 ====================

```
static int pci_conf1_read_config_word(struct pci_dev *dev, int where, u16 *value)
{
     outl(CONFIG_CMD(dev,where), 0xCF8);
     *value = inw(0xCFC + (where&2));
     return PCIBIOS_SUCCESSFUL;
}
```

这就是前面讲的通过 0xCF8 和 0xCFC 两个寄寸器访问配置寄存器的过程。至于另两个数据结构，我们在这里不感兴趣，但还是列出于下供有需要的读者参考（均定义于 arch/i386/kernel/pci-pc.c）。

==================== arch/i386/kernel/pci-pc.c 153 160 ====================

```
static struct pci_ops pci_direct_conf2 = {
     pci_conf2_read_config_byte,
     pci_conf2_read_config_word,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     pci_conf2_read_config_dword,
     pci_conf2_write_config_byte,
     pci_conf2_write_config_word,
     pci_conf2_write_config_dword
};
```

==================== arch/i386/kernel/pci-pc.c 531 542 ====================

```
/*
 * Function table for BIOS32 access
 */

static struct pci_ops pci_bios_access = {
     pci_bios_read_config_byte,
     pci_bios_read_config_word,
     pci_bios_read_config_dword,
     pci_bios_write_config_byte,
     pci_bios_write_config_word,
     pci_bios_write_config_dword
};
```

可见，如果要由 BIOS 完成对 PCI 设备上配置寄存器的读写，则通过一个函数 pci_bios_read_config_word()进入 BIOS，这个函数的代码也在同一文件（arch/i386/kernel/pci-pc.c）中：

==================== arch/i386/kernel/pci-pc.c 441 457 ====================

```
static int pci_bios_read_config_word(struct pci_dev *dev, int where, u16 *value)
{
     unsigned long ret;
     unsigned long bx = (dev->bus->number << 8) | dev->devfn;

     __asm__("lcall (%%esi); cld\n\t"
          "jc 1f\n\t"
          "xor %%ah, %%ah\n"
          "1:"
          : "=c" (*value),
            "=a" (ret)
          : "1" (PCIBIOS_READ_CONFIG_WORD),
            "b" (bx),
            "D" ((long) where),
            "S" (&pci_indirect));
     return (int) (ret & 0xff00) >> 8;
}
```

这段汇编代码通过一个指向目标地址和段地址的结构指针 pci_indirect 调用 BIOS 中的一个函数，这个函数就是 BIOS 为保护模式提供的 PCI 配置寄存器操作的总入口，其地址是通过扫描 PCI BIOS 的存储区间。根据若干特殊的字节（称为“签名”）而找到的。

除由 gcc 生成的上述这一组函数外，内核中还有另一组类似的、但是版本较老的函数 pcibios_read_config_byte()、pcibios_read_config_word()、pcibios_read_config_dword()、pcibios_write_config_byte()、pcibios_write_config_word()、pcibios_write_config_dword()，这是为了与老

版本的动态安装模块兼容而保留的，定义于 drivers/pci/compat.c：

==================== drivers/pci/compat.c 51 65 ====================

```
51  #define PCI_OP(rw,size,type)                              \
52  int pcibios_##rw##_config_##size (unsigned char bus, unsigned char dev_fn,\
53                      unsigned char where, unsigned type val) \
54  {                                              \
55      struct pci_dev *dev = pci_find_slot(bus, dev_fn);           \
56      if (!dev) return PCIBIOS_DEVICE_NOT_FOUND;                  \
57      return pci_##rw##_config_##size(dev, where, val);           \
58  }
59
60  PCI_OP(read, byte, char *)
61  PCI_OP(read, word, short *)
62  PCI_OP(read, dword, int *)
63  PCI_OP(write, byte, char)
64  PCI_OP(write, word, short)
65  PCI_OP(write, dword, int)
```

以 61 行为例，经过预处理以后就变成了这样：

```
int pcibios_read_config_word(unsigned char bus, unsigned char dev_fn,
   unsigned char where, unsigned type val)
{
   struct pci_dev *dev = pci_find_slot(bus, dev_fn);
   if (!dev) return PCIBIOS_DEVICE_NOT_FOUND;
   return pci_read_config_word(dev, where, val);
}
```

可见，最后还是“九九归一”，落实到新版本的 pci_read_config_word()上。

对 PCI 设备的枚举和配置都是在初始化阶段中完成的。一旦完成了初始化，以后的操作就比较简单了。PCI 总线的初始化由 pci_init()完成，其代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 1162 1175 ====================

```
void __init pci_init(void)
{
    struct pci_dev *dev;

    pcibios_init();

    pci_for_each_dev(dev) {
        pci_fixup_device(PCI_FIXUP_FINAL, dev);
    }

#ifdef CONFIG_PM
    pm_register(PM_PCI_DEV, 0, pci_pm_callback);
#endif
}
```

先提一下这里的条件编译选择项 CONFIG_PM，这是为电源管理而设的，PM 就是“Power Management”的意思，不过我们在这里对此不感兴趣。

一般而言，只要是采用 PCI 总线的 PC 机，其 BIOS 就必须提供对 PCI 总线操作的支持，因而称为 PCI BIOS。这种 BIOS 在机器加电以后的自检阶段会从系统中的第一个 PCI 桥，即“宿主—PCI 桥”开始进行探测和扫描，逐个地“枚举”（enumerate）连接在第一条 PCI 总线上的所有 PCI 设备并记录在案。如果其中的某个设备是个“PCI-PCI 桥”，则又前进一步，再探测和扫描连在这个桥上的次级 PCI 总线。就这样递归下去，直到穷尽系统中的所有 PCI 设备。其结果是在内存中建立起一棵代表着这些 PCI 总线和设备的“PCI 树”。一般的 PC 系统结构中都只有一个“宿主—PCI 桥”，但是如果有不止一个，也对其如法炮制。然后，操作系统可以通过 BIOS 调用来获取有关本系统中 PCI 设备的信息。当操作系统要进行对 PCI 设备的配置（config）操作时，也可以通过 BIOS 调用来完成。

但是，在实践中发现有些母板上的 PCI BIOS 有这样那样的问题，再说有些系统（如一些嵌入式系统）根本就没有 BIOS，所以后来 Linux 内核也提供了绕过 BIOS、直接探测和枚举 PCI 设备（以及配置）的功能，而把是否通过 BIOS 进行探测和枚举作为一个编译选择项。不过，尽管直接的 PCI 设备探测与枚举跟 BIOS 并无关系，代码的作者还是把这部分操作也放在 pcibios_init()中完成，以求跟老的函数名一致。

函数 pcibios_init()的代码在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 953 996 ====================
[pci_init()>pcibios_init()]

```
/*
 * Initialization. Try all known PCI access methods. Note that we support
 * using both PCI BIOS and direct access: in such cases, we use I/O ports
 * to access config space, but we still keep BIOS order of cards to be
 * compatible with 2.0.X. This should go away some day.
 */

void __init pcibios_init(void)
{
	struct pci_ops *bios = NULL;
	struct pci_ops *dir = NULL;

#ifdef CONFIG_PCI_BIOS
	if ((pci_probe & PCI_PROBE_BIOS) && ((bios = pci_find_bios()))) {
		pci_probe |= PCI_BIOS_SORT;
		pci_bios_present = 1;
	}
#endif
#ifdef CONFIG_PCI_DIRECT
	if (pci_probe & (PCI_PROBE_CONF1 | PCI_PROBE_CONF2))
		dir = pci_check_direct();
#endif
	if (dir)
		pci_root_ops = dir;
	else if (bios)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        pci_root_ops = bios;
    else {
        printk("PCI: No PCI bus detected\n");
        return;
    }

    printk("PCI: Probing PCI hardware\n");
    pci_root_bus = pci_scan_bus(0, pci_root_ops, NULL);

    pcibios_irq_init();
    pcibios_fixup_peer_bridges();
    pcibios_fixup_irqs();
    pcibios_resource_survey();

#ifdef CONFIG_PCI_BIOS
    if ((pci_probe & PCI_BIOS_SORT) && !(pci_probe & PCI_NO_SORT))
        pcibios_sort();
#endif
}
```

编译选择项 CONFIG_PCIBIOS 表示通过 BIOS 进行 PCI 设备的探测和枚举，CONFIG_PCI_DIRECT 表示直接进行 PCI 设备的探测和枚举。二者不是互斥的，也可以先试着通过 BIOS 探测，若不成功再亲自动手。我们对通过 PCI BIOS 进行的操作不感兴趣，但还是把 pci_find_bios()的代码列在下面供读者参考，一方面也让读者对怎样发现 PCI BIOS 并找到 PCI 配置操作的入口有个感性的认识。

==================== arch/i386/kernel/pci-pc.c 544 593 ====================

[pci_init()>pcibios_init()>pci_find_bios()]

```
/*
 * Try to find PCI BIOS.
 */

static struct pci_ops * __init pci_find_bios(void)
{
    union bios32 *check;
    unsigned char sum;
    int i, length;

    /*
     * Follow the standard procedure for locating the BIOS32 Service
     * directory by scanning the permissible address range from
     * 0xe0000 through 0xfffff for a valid BIOS32 structure.
     */

    for (check = (union bios32 *) __va(0xe0000);
         check <= (union bios32 *) __va(0xffff0);
         ++check) {
        if (check->fields.signature != BIOS32_SIGNATURE)
            continue;
```

```
565             length = check->fields.length * 16;
566             if (!length)
567                 continue;
568             sum = 0;
569             for (i = 0; i < length ; ++i)
570                 sum += check->chars[i];
571             if (sum != 0)
572                 continue;
573             if (check->fields.revision != 0) {
574                 printk("PCI: unsupported BIOS32 revision %d at 0x%p, report to <mj@suse.cz>\n",
575                     check->fields.revision, check);
576                 continue;
577             }
578             DBG("PCI: BIOS32 Service Directory structure at 0x%p\n", check);
579             if (check->fields.entry >= 0x100000) {
580                 printk("PCI: BIOS32 entry (0x%p) in high memory, cannot use.\n", check);
581                 return NULL;
582             } else {
583                 unsigned long bios32_entry = check->fields.entry;
584                 DBG("PCI: BIOS32 Service Directory entry at 0x%lx\n", bios32_entry);
585                 bios32_indirect.address = bios32_entry + PAGE_OFFSET;
586                 if (check_pcibios())
587                     return &pci_bios_access;
588             }
589             break;    /* Hopefully more than one BIOS32 cannot happen... */
590         }
591 
592         return NULL;
593     }
```

这里 BIOS32_SIGNATURE 是一串特殊的字节，定义于同一文件（arch/i386/kernel/pci-pc.c）中：

==================== arch/i386/kernel/pci-pc.c 259 260 ====================

```
259 /* BIOS32 signature: "_32_" */
260 #define BIOS32_SIGNATURE    (('_' << 0) + ('3' << 8) + ('2' << 16) + ('_' << 24))
```

我们的重点是对 PCI 设备的直接探测与枚举，这是由 pci_check_direct()完成的，其代码也在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 192 235 ====================

[pci_init()>pcibios_init()>pci_check_direct()]

```
192 static struct pci_ops * __init pci_check_direct(void)
193 {
194     unsigned int tmp;
195     unsigned long flags;
196 
197     __save_flags(flags); __cli();
198 
199     /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
200          * Check if configuration type 1 works.
201          */
202         if (pci_probe & PCI_PROBE_CONF1) {
203              outb (0x01, 0xCFB);
204              tmp = inl (0xCF8);
205              outl (0x80000000, 0xCF8);
206              if (inl (0xCF8) == 0x80000000 &&
207                  pci_sanity_check(&pci_direct_conf1)) {
208                   outl (tmp, 0xCF8);
209                   __restore_flags(flags);
210                   printk("PCI: Using configuration type 1\n");
211                   request_region(0xCF8, 8, "PCI conf1");
212                   return &pci_direct_conf1;
213              }
214              outl (tmp, 0xCF8);
215         }
216
217         /*
218          * Check if configuration type 2 works.
219          */
220         if (pci_probe & PCI_PROBE_CONF2) {
221              outb (0x00, 0xCFB);
222              outb (0x00, 0xCF8);
223              outb (0x00, 0xCFA);
224              if (inb (0xCF8) == 0x00 && inb (0xCFA) == 0x00 &&
225                  pci_sanity_check(&pci_direct_conf2)) {
226                   __restore_flags(flags);
227                   printk("PCI: Using configuration type 2\n");
228                   request_region(0xCF8, 4, "PCI conf2");
229                   return &pci_direct_conf2;
230              }
231         }
232
233         __restore_flags(flags);
234         return NULL;
235    }
```

如前所述，“宿主—PCI 桥”的 I/O 口一定在 I/O 地址为 0xCF8～0xCFF 处，其中 0xCF8～0xCFB 为地址口，0xCFC～0xCFF 为数据口。但是，也有可能系统中根本就没有 PCI 总线存在；或者再进一步，没有 PCI 总线存在，却凑巧有个 ISA 设备正在使用这些地址。针对这些可能性，PCI 总线标准规定了测试的方法，所以这里先按 1 型操作试试，如果成功就从 212 行返回了，不成功则再按 2 型试试。不过，如前所述，2 型操作现在已经不用了，所以对 1 型操作的测试一般总能成功。

探测到一个 1 型“宿主—PCI 桥”以后，要进一步采用 pci_direct_conf1 通过 pci_sanity_check()再加验证。这个函数的代码也在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 162 190 ====================

[pci_init()>pcibios_init()>pci_check_direct()>pci_sanity_check()]

```
162    /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Before we decide to use direct hardware access mechanisms, we try to do some
 * trivial checks to ensure it at least _seems_ to be working -- we just test
 * whether bus 00 contains a host bridge (this is similar to checking
 * techniques used in XFree86, but ours should be more reliable since we
 * attempt to make use of direct access hints provided by the PCI BIOS).
 *
 * This should be close to trivial, but it isn't, because there are buggy
 * chipsets (yes, you guessed it, by Intel and Compaq) that have no class ID.
 */
static int __init pci_sanity_check(struct pci_ops *o)
{
     u16 x;
     struct pci_bus bus;          /* Fake bus and device */
     struct pci_dev dev;

     if (pci_probe & PCI_NO_CHECKS)
          return 1;
     bus.number = 0;
     dev.bus = &bus;
     for(dev.devfn=0; dev.devfn < 0x100; dev.devfn++)
          if ((!o->read_word(&dev, PCI_CLASS_DEVICE, &x) &&
               (x == PCI_CLASS_BRIDGE_HOST || x == PCI_CLASS_DISPLAY_VGA)) ||
              (!o->read_word(&dev, PCI_VENDOR_ID, &x) &&
               (x == PCI_VENDOR_ID_INTEL || x == PCI_VENDOR_ID_COMPAQ)))
               return 1;
     DBG("PCI: Sanity check failed\n");
     return 0;
}
```

怎样验证呢？我们知道，一条 PCI 总线上最多可以有 256 种功能，或者说 256 个逻辑设备，而“宿主—PCI 桥”至少应该提供 PCI_CLASS_BRIDGE_HOST 和 PCI_CLASS_DISPLAY_VGA 这二者之一。如果都没有，那么至少这“宿主—PCI 桥”芯片应该是 Intel 或 Compaq 制造的。如果连这也得不到证实，那就还是只能认为“宿主—PCI 桥”实际上并不存在，前面从 PCI 口中读到的信息只是碰巧而已。

“宿主—PCI 桥”的存在得到验证以后，系统中就多了一项 I/O 设备，占据的 I/O 地址区间从 0xCF8 开始，长度为 8。所以，这里通过 request_region()在内核的 I/O 设备资源树（见后）增加一个节点。表示这个 I/O 地址区间已由“ PCI conf1”占用。

如果成功地发现了一个 1 型“宿主—PCI 桥”，函数最终返回指向 pci_direct_conf1 的指针。这个 pci_ops 数据结构的定义已经在前面看到过了。从此以后，在枚举和配置操作中采用什么方法访问 PCI 总线也就定下来了。

回到 pcibios_init()的代码中，接着（985 行）就是对 PCI 总线的扫描，即对连接在这条总线上的 PCI 设备的探测与枚举了，这是由 pci_scan_bus()完成的，其代码见 drivers/pci/pci.c：

==================== drivers/pci/pci.c 1045 1054 ====================

[pci_init()pcibios_init()>pci_scan_bus()]

```
struct pci_bus * __init pci_scan_bus(int bus, struct pci_ops *ops, void *sysdata)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct pci_bus *b = pci_alloc_primary_bus(bus);
    if (b) {
         b->sysdata = sysdata;
         b->ops = ops;
         b->subordinate = pci_do_scan_bus(b);
    }
    return b;
}
```

“宿主—PCI 桥”后面是系统中的第一条 PCI 总线，称为“主 PCI 总线”。在内核中，每条 PCI 总线都由一个 pci_bus 数据结构代表，所以先要为之分配一个数据结构，这种数据结构定义于 include/linux/pci.h 中：

==================== include/linux/pci.h 381 406 ====================

```
struct pci_bus {
    struct list_head node;      /* node in list of buses */
    struct pci_bus    *parent; /* parent bus this bridge is on */
    struct list_head children;  /* list of child buses */
    struct list_head devices;   /* list of devices on this bus */
    struct pci_dev    *self;        /* bridge device as seen by parent */
    struct resource   *resource[4]; /* address space routed to this bus */

    struct pci_ops    *ops;         /* configuration access functions */
    void     *sysdata;/* hook for sys-specific extension */
    struct proc_dir_entry *procdir; /* directory entry in /proc/bus/pci */

    unsigned char number;       /* bus number */
    unsigned char primary; /* number of primary bridge */
    unsigned char secondary;    /* number of secondary bridge */
    unsigned char subordinate;  /* max number of subordinate buses */

    char     name[48];
    unsigned short    vendor;
    unsigned short    device;
    unsigned int  serial;       /* serial number */
    unsigned char pnpver;       /* Plug & Play version */
    unsigned char productver;   /* product version */
    unsigned char checksum;/* if zero - checksum passed */
    unsigned char pad1;
};
```

系统中的每条 PCI 总线都有个编号 number，主 PCI 总线的编号为 0。

所有的 pci_bus 数据结构都互相连接在一起，形成若干（通常只有一棵）PCI 总线树，每棵树的根是一个代表着“宿主—PCI 桥”的 pci_bus 结构。内核中有一个队列头 pci_root_buses，所有代表着“宿主—PCI 桥”的 pci_bus 结构都通过其内部的队列头 node 挂在这个队列中。同时，每个 pci_bus 结构本身又维持着两个队列。一个是 devices，凡是连接在这条总线上的设备都有个 pci_dev 数据结构（见下）挂在这个队列中。另一个是 children，凡是通过“ PCI-PCI 桥”连接在这条总线上的次层 PCI 总线都有

PDF-XCHANGE Click to buy NOW! www.docu-track.com

个 pci-bus 数据结构挂在这个队列中。这样，从队列 pci_root_buses 开始的整个层次结构就反映着系统中 PCI 总线和设备的配备和连接。而 pci_scan_bus()的目的正是要在内存中建立起这样一个层次结构。通常系统中只有一个“宿主—PCI 桥”，所以 pci_root_buses 中通常只有一个节点，也就是只有一棵树。

每个 PCI 设备都由一个 pci_dev 数据结构代表，这种数据结构定义于 include/linux/pci.h：

==================== include/linux/pci.h 311 362 ====================

```
/*
 * The pci_dev structure is used to describe both PCI and ISAPnP devices.
 */
struct pci_dev {
	struct list_head global_list;	/* node in list of all PCI devices */
	struct list_head bus_list;	/* node in per-bus list */
	struct pci_bus	*bus;		/* bus this device is on */
	struct pci_bus	*subordinate;	/* bus this device bridges to */

	void		*sysdata;	/* hook for sys-specific extension */
	struct proc_dir_entry *procent;	/* device entry in /proc/bus/pci */

	unsigned int	devfn;		/* encoded device & function index */
	unsigned short	vendor;
	unsigned short	device;
	unsigned short	subsystem_vendor;
	unsigned short	subsystem_device;
	unsigned int	class;		/* 3 bytes: (base,sub,prog-if) */
	u8		hdr_type;	/* PCI header type (`multi' flag masked out) */
	u8		rom_base_reg;	/* which config register controls the ROM */

	struct pci_driver *driver;	/* which driver has allocated this device */
	void		*driver_data;	/* data private to the driver */
	dma_addr_t	dma_mask;	/* Mask of the bits of bus address this
					   device implements.  Normally this is
					   0xffffffff.  You only need to change
					   this if your device has broken DMA
					   or supports 64-bit transfers.  */

	/* device is compatible with these IDs */
	unsigned short vendor_compatible[DEVICE_COUNT_COMPATIBLE];
	unsigned short device_compatible[DEVICE_COUNT_COMPATIBLE];

	/*
	 * Instead of touching interrupt line and base address registers
	 * directly, use the values stored here. They might be different!
	 */
	unsigned int	irq;
	struct resource resource[DEVICE_COUNT_RESOURCE]; /* I/O and memory regions + expansion ROMs */
	struct resource dma_resource[DEVICE_COUNT_DMA];
	struct resource irq_resource[DEVICE_COUNT_IRQ];

```

```
	char		name[80];	/* device name */
	char		slot_name[8];	/* slot name */
	int		active;		/* ISAPnP: device is active */
	int		ro;		/* ISAPnP: read only */
	unsigned short	regs;		/* ISAPnP: supported registers */

	int (*prepare)(struct pci_dev *dev); /* ISAPnP hooks */
	int (*activate)(struct pci_dev *dev);
	int (*deactivate)(struct pci_dev *dev);
};
```

每个 pci_dev 数据结构都同时连入两个队列，一方面通过 global_list 挂入一个总的 pci_dev 结构队列；同时又通过 bus_list 挂入其所在总线的 pci_dev 结构队列 devices，并且使指针 bus 指向代表着该总线的 pci_bus 数据结构。如果具体的设备是个“ PCI-PCI 桥”则还要使其指针 subordinate 指向代表着另一个 PCI 总线（次层 PCI 总线）的 pci_bus 数据结构。

在 pci_bus 结构中有个 resource 数据结构数组，而 pci_dev 结构中更有三个这样的数组。每个 resource 数据结构都可以用来描述一个地址区间，包括其起点和终点。地址本身就是一种重要的“资源”；对于 PCI 总线的初始化，目的就在于为各个设备上的各个区间分配和设置地址，以代替从前手工设置跳线或小开关的过程，因而在这个过程中所涉及的首要资源就是地址，这种数据结构即因此而得名。从某种意义上说，PCI 总线的整个初始化过程就是与这些地址区间打交道。关于 resource 数据结构后面还要详细介绍。

随着代码的阅读，读者自会明白上面这两个数据结构中其它一些字段的作用。

函数 pci_alloc_primary_bus()的代码在 drivers/pci/pci.c 中：

===================== drivers/pci/pci.c 1026 1043 =====================

```
struct pci_bus * __init pci_alloc_primary_bus(int bus)
{
	struct pci_bus *b;

	if (pci_bus_exists(&pci_root_buses, bus)) {
		/* If we already got to this bus through a different bridge, ignore it */
		DBG("PCI: Bus %02x already known\n", bus);
		return NULL;
	}

	b = pci_alloc_bus();
	list_add_tail(&b->node, &pci_root_buses);

	b->number = b->secondary = bus;
	b->resource[0] = &ioport_resource;
	b->resource[1] = &iomem_resource;
	return b;
}
```

同一 PCI 总线只能由一个 pci_bus 结构代表，在 PCI 树中或 pci_root_buses 队列中只能出现一次，所以先要通过 pci_bus_exists()检查是否重复。参数 bus 为相应 PCI 总线的编号，在这里是 0（见前面的

PDF-XCHANGE Click to buy NOW! www.docu-track.com

985 行）。这里引用的 ioport_resource 和 iomem_resource 分别为反映着 I/O 空间和内存空间地址占用情况的两棵资源树，后面我们还要讲到。

为主 PCI 总线分配了数据结构以后，就可以开始扫描了。函数 pci_do_scan_bus()的代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 970 1012 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()]

```
970  static unsigned int __init pci_do_scan_bus(struct pci_bus *bus)
971  {
972       unsigned int devfn, max, pass;
973       struct list_head *ln;
974       struct pci_dev *dev, dev0;
975
976       DBG("Scanning bus %02x\n", bus->number);
977       max = bus->secondary;
978
979       /* Create a device template */
980       memset(&dev0, 0, sizeof(dev0));
981       dev0.bus = bus;
982       dev0.sysdata = bus->sysdata;
983
984       /* Go find them, Rover! */
985       for (devfn = 0; devfn < 0x100; devfn += 8) {
986            dev0.devfn = devfn;
987            pci_scan_slot(&dev0);
988       }
989
990       /*
991        * After performing arch-dependent fixup of the bus, look behind
992        * all PCI-to-PCI bridges on this bus.
993        */
994       DBG("Fixups for bus %02x\n", bus->number);
995       pcibios_fixup_bus(bus);
996       for (pass=0; pass < 2; pass++)
997            for (ln=bus->devices.next; ln != &bus->devices; ln=ln->next) {
998                 dev = pci_dev_b(ln);
999                 if (dev->hdr_type == PCI_HEADER_TYPE_BRIDGE || dev->hdr_type == PCI_HEADER_TYPE_CARDBUS)
1000                     max = pci_scan_bridge(bus, dev, max, pass);
1001           }
1002
1003      /*
1004       * We've scanned the bus and so we know all about what's on
1005       * the other side of any bridges that may be on this bus plus
1006       * any devices.
1007       *
1008       * Return how far we've got finding sub-buses.
1009       */
1010      DBG("Bus scan for %02x returning with max=%02x\n", bus->number, max);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    return max;
}
```

扫描一条 PCI 总线的直接目的就是逐个地发现连接在该总线上的 PCI 设备，为其建立起 pci_dev 数据结构并挂入相应的队列，这就是所谓枚举。所以，这里先准备下一个空白的 pci_dev 结构，然后依次对各个 PCI 接口通过 pci_scan_slot()扫描，每次扫描 8 个功能，即 8 个逻辑设备，这是每块 PCI 接口卡上的最大容量。注意这里的 devfn 是将前述的 5 位设备号与 3 位功能号合在一起的“逻辑设备号”。这个函数的定义也在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 932 968 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()]

```
struct pci_dev * __init pci_scan_slot(struct pci_dev *temp)
{
    struct pci_bus *bus = temp->bus;
    struct pci_dev *dev;
    struct pci_dev *first_dev = NULL;
    int func = 0;
    int is_multi = 0;
    u8 hdr_type;

    for (func = 0; func < 8; func++, temp->devfn++) {
        if (func && !is_multi)      /* not a multi-function device */
            continue;
        if (pci_read_config_byte(temp, PCI_HEADER_TYPE, &hdr_type))
            continue;
        temp->hdr_type = hdr_type & 0x7f;

        dev = pci_scan_device(temp);
        if (!dev)
            continue;
        pci_name_device(dev);
        if (!func) {
            is_multi = hdr_type & 0x80;
            first_dev = dev;
        }

        /*
         * Link the device to both the global PCI device chain and
         * the per-bus list of devices.
         */
        list_add_tail(&dev->global_list, &pci_devices);
        list_add_tail(&dev->bus_list, &bus->devices);

        /* Fix up broken headers */
        pci_fixup_device(PCI_FIXUP_HEADER, dev);
    }
    return first_dev;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

一个物理 PCI 设备（接口卡）可以是多功能的，也可以是单功能的，如果是单功能的则其逻辑设备号必是 8 的倍数（因为低 3 位的功能号为 0），而一个设备是否为多功能要从设备读入其头部类型以后才能知道（头部类型字节的最高位为 1 表示多功能），所以先假定为单功能。然后，就以 pci_bus 结构指针 temp 为参数通过 pci_read_config_byte()读设备的头部字节，注意这里的 temp->devfn 为目标逻辑设备号。如前所述，这个函数最后会通过由一个 pci_ops 结构提供的函数指针完成操作。读到了头部类型字节以后，其低 7 位为类型编码（见 946 行），接着就可以通过 pci_scan_device()进一步读取具体逻辑设备的配置信息了（drivers/pci/pci.c）。前面讲过，每项逻辑设备的配置寄存器组中有些信息是由厂商提供、固化在里面的，有些信息（如地址映射和总线号）则有待设置。

==================== drivers/pci/pci.c 898 930 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()]

```
898 /*
899  * Read the config data for a PCI device, sanity-check it
900  * and fill in the dev structure...
901  */
902 static struct pci_dev * __init pci_scan_device(struct pci_dev *temp)
903 {
904      struct pci_dev *dev;
905      u32 l;
906 
907      if (pci_read_config_dword(temp, PCI_VENDOR_ID, &l))
908           return NULL;
909 
910      /* some broken boards return 0 or ~0 if a slot is empty: */
911      if (l == 0xffffffff || l == 0x00000000 || l == 0x0000ffff || l == 0xffff0000)
912           return NULL;
913 
914      dev = kmalloc(sizeof(*dev), GFP_KERNEL);
915      if (!dev)
916           return NULL;
917 
918      memcpy(dev, temp, sizeof(*dev));
919      dev->vendor = l & 0xffff;
920      dev->device = (l >> 16) & 0xffff;
921 
922      /* Assume 32-bit PCI; let 64-bit PCI cards (which are far rarer)
923         set this higher, assuming the system even supports it.  */
924      dev->dma_mask = 0xffffffff;
925      if (pci_setup_device(dev) < 0) {
926           kfree(dev);
927           dev = NULL;
928      }
929      return dev;
930 }
```

首先读入第一个长字，其低 16 位为厂商编号，高 16 位为设备号。只要这些编号不是全 1 或全 0，就可以认为是有效的编号，从而设备存在。所以要分配一个新的 pci_dev 数据结构并设置有关的字段，

然后，通过 pci_setup_device()进一步从 PCI 口读入有关这个设备的信息，并继续设置这个数据结构（drivers/pci/pci.c）。

==================== drivers/pci/pci.c 841 896 ====================
[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()>pci_setup_device()]

```
/*
 * Fill in class and map information of a device
 */
int pci_setup_device(struct pci_dev * dev)
{
     u32 class;

     sprintf(dev->slot_name, "%02x:%02x.%d", dev->bus->number, PCI_SLOT(dev->devfn), PCI_FUNC(dev->devfn));
     sprintf(dev->name, "PCI device %04x:%04x", dev->vendor, dev->device);

     pci_read_config_dword(dev, PCI_CLASS_REVISION, &class);
     class >>= 8;                    /* upper 3 bytes */
     dev->class = class;
     class >>= 8;

     DBG("Found %02x:%02x [%04x/%04x] %06x %02x\n", dev->bus->number, dev->devfn, dev->vendor, dev->device, class, dev->hdr_type);

     switch (dev->hdr_type) {             /* header type */
     case PCI_HEADER_TYPE_NORMAL:              /* standard header */
          if (class == PCI_CLASS_BRIDGE_PCI)
               goto bad;
          pci_read_irq(dev);
          pci_read_bases(dev, 6, PCI_ROM_ADDRESS);
          pci_read_config_word(dev, PCI_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
          pci_read_config_word(dev, PCI_SUBSYSTEM_ID, &dev->subsystem_device);
          break;

     case PCI_HEADER_TYPE_BRIDGE:              /* bridge header */
          if (class != PCI_CLASS_BRIDGE_PCI)
               goto bad;
          pci_read_bases(dev, 2, PCI_ROM_ADDRESS1);
          break;

     case PCI_HEADER_TYPE_CARDBUS:             /* CardBus bridge header */
          if (class != PCI_CLASS_BRIDGE_CARDBUS)
               goto bad;
          pci_read_irq(dev);
          pci_read_bases(dev, 1, 0);
          pci_read_config_word(dev, PCI_CB_SUBSYSTEM_VENDOR_ID, &dev->subsystem_vendor);
          pci_read_config_word(dev, PCI_CB_SUBSYSTEM_ID, &dev->subsystem_device);
          break;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
883      default:                    /* unknown header */
884          printk(KERN_ERR "PCI: device %s has unknown header type %02x, ignoring.\n",
885              dev->slot_name, dev->hdr_type);
886          return -1;
887
888      bad:
889          printk(KERN_ERR "PCI: %s: class %x doesn't match header type %02x. Ignoring class.\n",
890                dev->slot_name, class, dev->hdr_type);
891          dev->class = PCI_CLASS_NOT_DEFINED;
892      }
893
894      /* We found a fine healthy device, go go go... */
895      return 0;
896  }
```

接着是读入用于设备类别和版本号的长字。再往下就取决于具体设备的头部类型了（858 行）。头部类型 PCI_HEADER_TYPE_NORMAL 表示该设备为一般的 PCI 设备，PCI_HEADER_TYPE_BRIDGE 表示“ PCI-PCI 桥”，PCI_HEADER_TYPE_CARDBUS 则表示“ PCI-CardBus 桥”。

先看一般的 PCI 设备。PCI 设备通常都是可以发出中断请求的，所以设备配置寄存器组中有两个字节，即 PCI_INTERRUPT_PIN 和 PCI_INTERRUPT_LINE，反映着该设备的中断请求信号线与总线和系统的连接方式。在 PCI 总线上有 INTA～INTD 共 4 条中断请求线，从而在 PCI 插槽中有 4 根“针”。并非所有的 PCI 设备都能产生中断请求，例如图形卡就多半不能产生中断请求。如果 PCI_INTERRUPT_PIN 字节为 0，就表示本设备不能产生中断请求。但是，如果一个 PCI 设备能产生中断请求，那么在设备内部必定已经把中断请求连到 PCI 总线的某条中断请求线上，此时 PCI_INTERRUPT_PIN 字节的数值（1～4）表示该设备的中断请求连在哪一条线上。这种连接是由硬件决定的，所以 PCI_INTERRUPT_PIN 字节是个只读的字节，不能通过软件设置。可是，连在哪一条 PCI 中断请求线上只是事情的一个方面，还有个最终是连接到系统的中断控制器（8259A 或 APIC）上的哪一条中断请求线的问题，称为“中断请求路径”，这就是以前需要通过跳线或小开关设置的两个内容之一。这是由软件选择和设置的，选择的结果就存储在 PCI_INTERRUPT_LINE 字节中。这里要指出，这个寄存器的目的只是保存信息，而并不带有控制功能。所以如果把这个寄存器的内容从 8 改成 9 并不意味着改变了连接的目标。

这里通过 pci_read_irq() 读入这两个字节，并把中断请求线号记录在 pei_dev 结构中（drivers/pci/pci.c）。

==================== drivers/pci/pci.c 827 839 ====================
[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()>pci_setup_device()>pci_read_irq()]

```
827  /*
828   * Read interrupt line and base address registers.
829   * The architecture-dependent code can tweak these, of course.
830   */
831  static void pci_read_irq(struct pci_dev *dev)
832  {
833      unsigned char irq;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &irq);
    if (irq)
        pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &irq);
    dev->irq = irq;
}
```

PCI 设备（接口）中一般都带有一些 RAM 和 ROM 区间，通常的控制 / 状态寄存器和数据寄存器也往往以 RAM 区间的形式出现，这些地址以后都要先映射到系统总线上，再进一步映射到内核的虚存空间。现在先要通过 pci_read_bases()把这些区间的大小和当前的地址读进来。对于一般的 PCI 设备，最多可以有 6 个这样的 RAM 区间。函数 pci_read_bases()的代码在 drivers/pci/pci.c 中，我们分两段阅读。

==================== drivers/pci/pci.c 547 579 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()>pci_setup_device()>pci_read_bases()]

```
static void pci_read_bases(struct pci_dev *dev, unsigned int howmany, int rom)
{
    unsigned int pos, reg, next;
    u32 l, sz;
    struct resource *res;

    for(pos=0; pos<howmany; pos = next) {
        next = pos+1;
        res = &dev->resource[pos];
        res->name = dev->name;
        reg = PCI_BASE_ADDRESS_0 + (pos << 2);
        pci_read_config_dword(dev, reg, &l);
        pci_write_config_dword(dev, reg, ~0);
        pci_read_config_dword(dev, reg, &sz);
        pci_write_config_dword(dev, reg, l);
        if (!sz || sz == 0xffffffff)
            continue;
        if (l == 0xffffffff)
            l = 0;
        if ((l & PCI_BASE_ADDRESS_SPACE) == PCI_BASE_ADDRESS_SPACE_MEMORY) {
            res->start = l & PCI_BASE_ADDRESS_MEM_MASK;
            sz = pci_size(sz, PCI_BASE_ADDRESS_MEM_MASK);
        } else {
            res->start = l & PCI_BASE_ADDRESS_IO_MASK;
            sz = pci_size(sz, PCI_BASE_ADDRESS_IO_MASK & 0xffff);
        }
        res->end = res->start + (unsigned long) sz;
        res->flags |= (l & 0xf) | pci_calc_resource_flags(l);
        if ((l & (PCI_BASE_ADDRESS_SPACE | PCI_BASE_ADDRESS_MEM_TYPE_MASK))
           == (PCI_BASE_ADDRESS_SPACE_MEMORY | PCI_BASE_ADDRESS_MEM_TYPE_64)) {
            pci_read_config_dword(dev, reg+4, &l);
            next++;
#if BITS_PER_LONG == 64
```

. . . . . .

==================== drivers/pci/pci.c 588 597 ====================

```
588 #else
589             if (l) {
590                 printk(KERN_ERR "PCI: Unable to handle 64-bit address for device %s\n", 
dev->slot_name);
591                 res->start = 0;
592                 res->flags = 0;
593                 continue;
594             }
595 #endif
596         }
597     }
```

这里的 dev->resource[]是 pci_dev 数据结构中的一个 resource 结构数组，用来记录设备上各个地址区间的起始地址与长度。数组的大小为 12，这是因为除 PCI 设备的 6 个常规地址区间以外还可以有一个扩充 ROM 区间，同时还要考虑到设备为 PCI 桥时的需要。

在设备的配置寄存器组中，用于 6 个常规地址区间的长字都是连续的，所以可以通过一个 for 循环来读出，代码中的 557 行计算出各个长字的起始地址。操作时，先从相应的长字中读（558 行），读得的数值是区间的（本地）起始地址与其它一些信息的组合，对这个数值的解释是：

- 首先，其最低位，即 bit0 表示区间的类型，为 0 表示是个存储器区间，为 1 则表示是个 I/O 地址（即寄存器）区间。注意这只是说可以通过什么样的操作（I/O 指令或访内指令）来访问这个区间，而与其内容是否为寄存器无关，寄存器也可以通过存储单元的形式来实现（称为“ memory mapped”）。
- 如果是存储器区间，那么其高 28 位就是起始地址的高 28 位（起始地址的最低 4 位一定是 0）。
- 如果是 I/O 区间，那么其高 29 位就是起始地址的高 29 位（起始地址的最低 3 位一定是 0）。
- 对于存储器区间，如果 bit3 为 1，就表示对这个区间的操作可以流水线化，称为“可预取 ”（prefetchable），否则就不能流水线化，而只能一个单元一个单元地读写（见后）。
- 还有，bit2 为 1 表示采用 64 位地址，为 0 表示采用 32 位地址。
- 最后，bit1 为 1 表示区间的大小超过 1MB，为 0 则表示区间的大小在 1MB 以下。

下列的一些常数定义反映了对这些标志位的约定（include/linux/pci.h）：

==================== include/linux/pci.h 84 91 ====================

```
#define  PCI_BASE_ADDRESS_SPACE  0x01 /* 0 = memory, 1 = I/O */
#define  PCI_BASE_ADDRESS_SPACE_IO 0x01
#define  PCI_BASE_ADDRESS_SPACE_MEMORY 0x00
#define  PCI_BASE_ADDRESS_MEM_TYPE_MASK 0x06
#define  PCI_BASE_ADDRESS_MEM_TYPE_32 0x00 /* 32 bit address */
#define  PCI_BASE_ADDRESS_MEM_TYPE_1M 0x02 /* Below 1M [obsolete] */
#define  PCI_BASE_ADDRESS_MEM_TYPE_64 0x04 /* 64 bit address */
#define  PCI_BASE_ADDRESS_MEM_PREFETCH    0x08 /* prefetchable? */
```

从配置寄存器组的一个长字中读出了区间的起始地址以后，往同一长字中写入全 1（559 行），即 0xffffffff，接着再从同一长字中读（560 行），这时候读得的数值便是区间的大小。这个数值的低 4 位

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

或低 3 位为控制信息，这一点上与起始地址的格式相似，但是在其高 28 位或 29 位中只有位置最低的那位 1 才有效。区间的大小必定是 2 的某次幂，所以其二进制数值中应该只有一位是 1，而其它各位均为 0。但是，在读得的数值中却通常有多位是 1，此时只有位置最低的那个 1 才有效，所以要通过 pci_size() 加以换算（drivers/pci/pci.c）。

==================== drivers/pci/pci.c 537 545 ====================
[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()>pci_setup_device()>pci_read_bases()>pci_size()]

```
/*
 * Find the extent of a PCI decode..
 */
static u32 pci_size(u32 base, unsigned long mask)
{
	u32 size = mask & base;		/* Find the significant bits */
	size = size & ~(size-1);	/* Get the lowest of them to find the decode size */
	return size-1;			/* extent = size - 1 */
}
```

这里先用 mask 把最低的 4 位或 3 位屏蔽掉，然后再把位置最低的那个 1 抽取出来。例如，假定把最低的 4 位或 3 位屏蔽掉以后得到 size 的数值是 0xffff0100，则(size-1)为 0xffff00ff，而-(size-1)就是 0x0000ff00，最后得到 size = 0xffff0100 & 0x0000ff00 = 0x100。这样，就把 size 的二进制数值中位置最低的那个 1 抽取了出来，从而得到 size 的实际数值为 0x100，即区间的大小为 256，而返回的值则为 255。读者一定会问，为什么不直接就读回 0x100，而要搞得这么复杂呢？当然，前面那些为 1 的位是有意义的。事实上，除位置最低的那个 1 表示着区间的大小以外，所有为 l 的位（包括位置最低的 1）都是可以（在建立地址映射时）加以设置的，而所有为 0 的位则不能设置。在这个例子中，就表示该区间在映射后的起始地址必须是 64KB 边界上的第一个或第二个 256 字节。我们在这里为了说明问题而举了一个比较特殊的例子，多数情况下从位置最低的 1 开始往上所有各位都是 1，表示区间的起点是与区间的大小“自然”对齐的。

读取了区间大小以后，还要把起始地址写回这个长字（561 行），恢复其原状。

然后，把从寄存器读出的内容换算成区间的起点和终点，记录在相应的 resource 结构中。

对“可预取”这个概念可能还需要一些说明。如上所述，一个区间之为“存储器区间”或“I/O 区间”取决于该区间的地址是在“存储器地址空间”或“I/O 地址空间”，实质上取决于 CPU 通过哪一类的指令访问这个区间，而与在具体地址上的是存储单元或寄存器无关。事实上，有些 CPU 根本就没有 I/O 指令，因而就没有 I/O 地址空间，所有的寄存器都在存储器地址空间中。另一方面，以前也讲过，i386 处理器虽有 I/O 地址空间，但是其 I/O 地址空间比较小（16 位），因而比较拥挤。再说通过 I/O 指令访问就意味着每次访问时都得调用一个汇编语言子程序（因为 C 语言中没有相应的语言成分），而不像通过访内指令时那样可以把寄存器看作一个变量，因而不那么方便，效率也要低一些。所以，在 PCI 设备中一般都倾向于将寄存器映射到存储器地址空间，而避免使用 I/O 地址空间。然而，即使都使用存储器地址空间，寄存器与真正的存储单元还是有着本质的不同，主要在于从一个单元读出时是否可能改变其内容。对于普通的存储单元，读操作是不会改变其内容的，所以反复从同一单元读出多次也没有关系，每次读出的内容都一样。而寄存器就不同了。首先，有些寄存器可能代表着一个 FIFO 队列，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

从中读出时每次都是读出 FIFO 中最前面的内容，因而每次都可能不同。其次，很多状态寄存器中的状态位在读出时就清成了 0。不过，二者间的这种区别对于个别的读操作，即每次只从一个单元的读出并无影响。但是，对于连续的、成块的读出就有影响了。为了提高成块读出的效率，PCI 总线对成块的读出加以流水线化，在应 CPU 的要求读出单元 N 后预先就把 N+1 也读入流水线，这样当 CPU 真的接着要求读出单元 N+1 时就不用等待了，这就是所谓“预取”。显然，预取是建立在对 CPU 意图的猜测上的，这种猜测有可能（通常）成真，也有可能失败。如果猜测失败了，预取的内容就丢掉了，对于存储单元这并无害处，下次需要时再读就是了。可是对寄存器就不同了，下次再读时很可能其内容已经因为预取而变化了。所以，一般而言，映射在存储器地址空间中的寄存器是不可预取的。

我们在这里不关心 64 位 PCI 总线设备，即采用 64 位地址的设备，所以跳过对此的检查和处理（575～596 行）。

除常规的 6 个存储区间外，设备上还可能提供一个扩充 ROM 区间，用于这个区间的长字与前面这 6 个不连在一起，所以要放在 for 循环外面单独处理。不过，所作的处理与前面并无大的不同，我们把下面这段代码留给读者。

==================== drivers/pci/pci.c 598 616 ====================
[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_scan_device()>pci_setup_device()>pci_read_bases()]

```
	if (rom) {
		dev->rom_base_reg = rom;
		res = &dev->resource[PCI_ROM_RESOURCE];
		pci_read_config_dword(dev, rom, &l);
		pci_write_config_dword(dev, rom, ~PCI_ROM_ADDRESS_ENABLE);
		pci_read_config_dword(dev, rom, &sz);
		pci_write_config_dword(dev, rom, l);
		if (l == 0xffffffff)
			l = 0;
		if (sz && sz != 0xffffffff) {
			res->flags = (l & PCI_ROM_ADDRESS_ENABLE) |
			  IORESOURCE_MEM | IORESOURCE_PREFETCH | IORESOURCE_READONLY | IORESOURCE_CACHEABLE;
			res->start = l & PCI_ROM_ADDRESS_MASK;
			sz = pci_size(sz, PCI_ROM_ADDRESS_MASK);
			res->end = res->start + (unsigned long) sz;
		}
		res->name = dev->name;
	}
}
```

对 ROM 区间地址的解读与 RAM 区间的有所不同，其最低的 11 位都是保留的，其中 bit0 为 PCI_ROM_ADDRESS_ENABLE，为 1 时表示区间有效。

回到 pci_setup_device()的代码中（864 行），还要把设备的子系统号和子系统厂商号也读进来。

这里对“ PCI-PCI 桥”的处理（869～872 行）就比较简单了，许多字段，如中断请求等等，对于“ PCI-PCI 桥”都是无意义的。所以对此只需要调用 pci_read_bases()就可以了。同时，“ PCI-PCI 桥 ”上可以有的存储区间数量也少，最多只能有两个。当然，实际上对“ PCI-PCI 桥”的处理更为复杂，因为还得进一步扫描次层 PCI 总线，不过这是以后的事。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

至此，对 pci_scan_device()的调用已经完成，回到了 pci_scan_slot()中，返回的是一个 pci_dev 结构指针，如果该指针为 0 则表示 PCI 总线上没有所指定的逻辑设备。接着的 pci_name_device()将从设备中读入的厂商编号和设备编号转换成字符串。内核中为此设置了一个小小的数据库（转换表），其内容见 drivers/pci/pci.ids，这里略举其中数行让读者有个印象：

==================== drivers/pci/pci.ids 1469 1472 ====================

```
10b7  3Com Corporation
	0001  3c985 1000BaseSX
	3390  Token Link Velocity
	3590  3c359 TokenLink Velocity XL
```

意思是厂商编号 0x10b7 代表着 3Com，而若设备编号为 0x0001 则表示这是该公司生产的 3c985 网络卡。这个文件中包括了几乎所有知名的厂商和它们的产品。

接着，把 pci_scan_device()返回的 pci_dev 数据结构挂入总的队列 pci_devices 以及该设备所在总线的队列。前面讲过，这个队列的头在代表着这个总线的 pci_bus 数据结构中。

本来，从一个逻辑设备读出头部信息并为之建立相应的数据结构以后，对这个设备的枚举就完成了。如前所述，这些信息是固化在芯片中的。可是，有些厂商在开始出售它们的某些产品以后却发现固化在里面的信息有错，因而需要在读出这些信息以后通过软件手段加以修正。有的修正需要在从设备读出头部信息后进行，有的则需要在设置了各个区间的总线地址以后进行。而厂商，则在出售产品时随同提供一段相应的程序。这样的情况不是一个两个，而是有许多。为此目的，Linux 内核中设计了一种统一的机制来进行这样的修正。首先是个数据结构（类型）pci_fixup，定义于 include/linux/pci.h：

==================== include/linux/pci.h 660 671 ====================

```
/*
 *  The world is not perfect and supplies us with broken PCI devices.
 *  For at least a part of these bugs we need a work-around, so both
 *  generic (drivers/pci/quirks.c) and per-architecture code can define
 *  fixup hooks to be called for particular buggy devices.
 */

struct pci_fixup {
	int pass;
	u16 vendor, device;			/* You can use PCI_ANY_ID here of course */
	void (*hook)(struct pci_dev *dev);
};
```

这个数据结构表示：对于由厂商 vendor 提供的设备 device，需要在什么时候（pass）执行通过函数指针 hook 提供的函数。内核中有两个 pci_fixup 结构数组，提供了对已知厂商和产品的修正手段，分别定义于 arch/i386/kernel/pci-pc.c 和 drivers/pci/quirks.c 中。为篇幅的原因，我们从这两个结构数组中删去了许多元素，只留下几个，让读者有个印象。

==================== arch/i386/kernel/pci-pc.c 927 929 ====================

```
struct pci_fixup pcibios_fixups[] = {
	{ PCI_FIXUP_HEADER,	PCI_VENDOR_ID_INTEL,	PCI_DEVICE_ID_INTEL_82451NX,
pci_fixup_i450nx },
```

```
929      { PCI_FIXUP_HEADER,    PCI_VENDOR_ID_INTEL,   PCI_DEVICE_ID_INTEL_82454GX,
     pci_fixup_i450gx },
     . . . . . .
```

==================== arch/i386/kernel/pci-pc.c 938 940 ====================

```
938      { PCI_FIXUP_HEADER,    PCI_VENDOR_ID_SI,  PCI_DEVICE_ID_SI_5598,      pci_fixup_latency },
939      { 0 }
940  };
```

==================== drivers/pci/quirks.c 250 255 ====================

```
250  /*
251   *  The main table of quirks.
252   */
253
254  static struct pci_fixup pci_fixups[] __initdata = {
255      { PCI_FIXUP_FINAL, PCI_VENDOR_ID_INTEL,   PCI_DEVICE_ID_INTEL_82441,  quirk_passive_release },
     . . . . . .
```

==================== drivers/pci/quirks.c 283 288 ====================

```
283      { PCI_FIXUP_HEADER,    PCI_VENDOR_ID_AL,  PCI_DEVICE_ID_AL_M7101,     quirk_ali7101_acpi },
284      { PCI_FIXUP_HEADER,    PCI_VENDOR_ID_INTEL,   PCI_DEVICE_ID_INTEL_82371SB_2,
     quirk_piix3_usb },
285      { PCI_FIXUP_HEADER,    PCI_VENDOR_ID_INTEL,   PCI_DEVICE_ID_INTEL_82371AB_2,
     quirk_piix3_usb },
286      { 0 }
287  };
288
```

例如，928 行说明：对于 Intel 的 82451NX 芯片（是“宿主—PCI 桥”）需要在读出了头部信息以后调用 pci_fixup_i450nx()进行修正（PCI_FIXUP_HEADER 在 include/linux/pci.h 中定义为 1，PCI_FIXUP_FINAL 则为 2）。具体的修正是由 pci_fixup_device()完成的，其代码在 drivers/pci/quirks.c 中，注意这里的调用参数 pass 为 PCI_FIXUP_HEADER，表示是读出头部信息以后的修正。

==================== drivers/pci/quirks.c 305 309 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_fixup_device()]

```
305  void pci_fixup_device(int pass, struct pci_dev *dev)
306  {
307      pci_do_fixups(dev, pass, pcibios_fixups);
308      pci_do_fixups(dev, pass, pci_fixups);
309  }
```

这个函数对上述两个数组分别调用 pci_do_fixups()，其代码在同一文件（drivers/pci/quirks.c）中：

==================== drivers/pci/quirks.c 290 303 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_slot()>pci_fixup_device()>pci_do_fixups()]

```
290  static void pci_do_fixups(struct pci_dev *dev, int pass, struct pci_fixup *f)
291  {
292      while (f->pass) {
293          if (f->pass == pass &&
294              (f->vendor == dev->vendor || f->vendor == (u16) PCI_ANY_ID) &&
295              (f->device == dev->device || f->device == (u16) PCI_ANY_ID)) {
```

```
296 #ifdef DEBUG
297             printk("PCI: Calling quirk %p for %s\n", f->hook, dev->slot_name);
298 #endif
299             f->hook(dev);
300         }
301         f++;
302     }
303 }
```

注意 293 行的条件，修正函数仅在指定的 pass 中才会执行。至于具体的修正函数，我们就不看了。

回到 pci_do_scan_bus()的代码中，完成了对 pci_scan_slot()的循环调用后，对一条 PCI 总线的扫描与枚举原则上已经完成了。但是，对得到的信息也可能需要作一些调整和修正。函数 pcibios_fixup_bus()的代码在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 942 951 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pcibios_fixup_bus()]

```
/*
 *  Called after each bus is probed, but before its children
 *  are examined.
 */

void __init pcibios_fixup_bus(struct pci_bus *b)
{
    pcibios_fixup_ghosts(b);
    pci_read_bridge_bases(b);
}
```

为什么要有这种调整和修正呢？原来，不光是 PCI 芯片中的信息会有错误，有些系统母板的设计也会发生错误。实践中发现，在有些母板上会在两个不同的综合地址（总是相距 16）中读出同一设备的头部信息，从而在枚举的过程中造成重复枚举，在所形成的 PCI 树中引入了“幻影”。所以，要通过 pcibios_fixup_ghosts()进行一次扫描，把“幻影”去掉。这个函数的代码在 arch/i386/kernel/pci-pc.c 中，我们在这里就不看了。

函数 pci_read_bridge_bases()的代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 618 697 ====================

[pci_init()>pcibios_init()>pci_scan_bus()>pci_do_scan_bus()>pcibios_fixup_bus()>pci_read_bridge_bases()]

```
void __init pci_read_bridge_bases(struct pci_bus *child)
{
    struct pci_dev *dev = child->self;
    u8 io_base_lo, io_limit_lo;
    u16 mem_base_lo, mem_limit_lo, io_base_hi, io_limit_hi;
    u32 mem_base_hi, mem_limit_hi;
    unsigned long base, limit;
    struct resource *res;
    int i;

    if (!dev)     /* It's a host bus, nothing to read */
```

```
		return;

	for(i=0; i<3; i++)
		child->resource[i] = &dev->resource[PCI_BRIDGE_RESOURCES+i];

	res = child->resource[0];
	pci_read_config_byte(dev, PCI_IO_BASE, &io_base_lo);
	pci_read_config_byte(dev, PCI_IO_LIMIT, &io_limit_lo);
	pci_read_config_word(dev, PCI_IO_BASE_UPPER16, &io_base_hi);
	pci_read_config_word(dev, PCI_IO_LIMIT_UPPER16, &io_limit_hi);
	base = ((io_base_lo & PCI_IO_RANGE_MASK) << 8) | (io_base_hi << 16);
	limit = ((io_limit_lo & PCI_IO_RANGE_MASK) << 8) | (io_limit_hi << 16);
	if (base && base <= limit) {
		res->flags = (io_base_lo & PCI_IO_RANGE_TYPE_MASK) | IORESOURCE_IO;
		res->start = base;
		res->end = limit + 0xfff;
		res->name = child->name;
	} else {
		/*
		 * Ugh. We don't know enough about this bridge. Just assume
		 * that it's entirely transparent.
		 */
		printk("Unknown bridge resource %d: assuming transparent\n", 0);
		child->resource[0] = child->parent->resource[0];
	}

	res = child->resource[1];
	pci_read_config_word(dev, PCI_MEMORY_BASE, &mem_base_lo);
	pci_read_config_word(dev, PCI_MEMORY_LIMIT, &mem_limit_lo);
	base = (mem_base_lo & PCI_MEMORY_RANGE_MASK) << 16;
	limit = (mem_limit_lo & PCI_MEMORY_RANGE_MASK) << 16;
	if (base && base <= limit) {
		res->flags = (mem_base_lo & PCI_MEMORY_RANGE_TYPE_MASK) | IORESOURCE_MEM;
		res->start = base;
		res->end = limit + 0xfffff;
		res->name = child->name;
	} else {
		/* See comment above. Same thing */
		printk("Unknown bridge resource %d: assuming transparent\n", 1);
		child->resource[1] = child->parent->resource[1];
	}

	res = child->resource[2];
	pci_read_config_word(dev, PCI_PREF_MEMORY_BASE, &mem_base_lo);
	pci_read_config_word(dev, PCI_PREF_MEMORY_LIMIT, &mem_limit_lo);
	pci_read_config_dword(dev, PCI_PREF_BASE_UPPER32, &mem_base_hi);
	pci_read_config_dword(dev, PCI_PREF_LIMIT_UPPER32, &mem_limit_hi);
	base = (mem_base_lo & PCI_MEMORY_RANGE_MASK) << 16;
	limit = (mem_limit_lo & PCI_MEMORY_RANGE_MASK) << 16;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#if BITS_PER_LONG == 64
     base |= ((long) mem_base_hi) << 32;
     limit |= ((long) mem_limit_hi) << 32;
#else
     if (mem_base_hi || mem_limit_hi) {
          printk(KERN_ERR "PCI: Unable to handle 64-bit address space for %s\n", child->name);
          return;
     }
#endif
     if (base && base <= limit) {
          res->flags = (mem_base_lo & PCI_MEMORY_RANGE_TYPE_MASK) | IORESOURCE_MEM | IORESOURCE_PREFETCH;
          res->start = base;
          res->end = limit + 0xfffff;
          res->name = child->name;
     } else {
          /* See comments above */
          printk("Unknown bridge resource %d: assuming transparent\n", 2);
          child->resource[2] = child->parent->resource[2];
     }
}
```

PCI 桥的配置寄存器组与一般 PCI 设备的不同。如前所述，一般 PCI 设备可以有 6 个地址区间，外加一个 ROM 区间，代表着设备上实际存在的存储器或寄存器区间；而 PCI 桥，则本身并不一定有存储器或寄存器区间，但是却有三个用于地址过滤的区间。每个地址过滤区间决定了一个地址窗口，从靠近 CPU 一侧发出的地址，如果落在 PCI 桥的某个窗口之内，就可以穿过 PCI 桥而到达其所连接的总线上。反过来，从总线一侧由 PCI 设备发出的地址（如果有的话），则要在相应 PCI 桥的所有窗口之外才能穿过 PCI 桥到达靠近 CPU 一侧。此外，PCI 桥的命令寄存器中还有“ memory access enable”和“ I/O access enable”两个控制位，当这两个控制位为 0 时，这些窗口就全部关上了（两个方向都关上）。在未完成对 PCI 总线的初始化之前，还没有为 PCI 设备上的各个区间分配合适的总线地址时，正是因为这两个控制位为 0，才不会对 CPU 一侧造成干扰。

显然，每个窗口的大小和位置应该是总线上各个相应区间的总和，而总线上的各个区间则应该基本上互相连续并且互不重叠。因此，一个窗口实际上也是一个区间，所以也用一个 resource 数据结构表示。不过，说是各个相应区间的总和，实际上窗口一定是按某种边界对齐的，其大小可能会大于实际的需要（从而造成少量浪费）。

第一个区间是 I/O 地址的窗口。PCI 桥上有两个 8 位寄存器，即 PCI_IO_BASE 和 PCI_IO_LIMIT，用来确定具体窗口的起点和终点。其中高 4 位就是 16 位 I/O 地址中的最高 4 位，低 4 位则固定为 0（对 16 位 I/O 地址而言，见下）。窗口的大小为 4KB 的倍数，并与 4KB 边界对齐。其起点为 PCI_IO_BASE 的高 4 位后面添上 12 位 0，终点则为 PCI_IO_LIMIT 的高 4 位后面添上 12 位 1。例如，要是 PCI_IO_BASE 的高 4 位为 6 而 PCI_IO_LIMIT 的高 4 位为 7，就表示窗口的范围为 0x6000 至 0x7fff。PCI 总线并不是专为 i386 处理器设计的，有些处理器可能使用 32 位 I/O 地址空间。所以，如果具体的 PCI 设备支持 32 位 I/O 地址，则进一步通过 PCI_IO_BASE_UPPER16 和 PCI_IO_LIMIT_UPPER16 两个 16 位寄存器提供窗口起点和终点的高 16 位，并且使 PCI_IO_BASE 和 PCI_IO_LIMIT 中的低 4 位固定为 1。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第二个区间是存储器地址的窗口。寄存器 PCI_MEMORY_BASE 和 PCI_MEMORY_LIMIT 的构造与作用跟上述 PCI_IO_BASE 和 PCI_IO_LIMIT 相似，只不过是 16 位寄存器，除最低 4 位为 0 以外，其高 12 位为 32 位存储器地址中的最高 12 位。窗口的大小则为 1MB 的倍数，并与 1MB 边界对齐。例如，要是 PCI_MEMORY_BASE 的高 12 位为 0xa81 而 PCI_MEMORY_LIMIT 的高 12 位也是 0xa81，就表示窗口的范围为 0xa8100000～0xa81fffff，共 1MB。这个区间主要用于映射在存储器地址空间的 I/O 寄存器。

第三个区间是“可预取”存储器地址的窗口。寄存器 PCI_PREF_MEMORY_BASE 和 PCI_PREF_MEMORY_LIMIT 的构造与作用跟 PCI_MEMORY_BASE 和 PCI_MEMORY_LIMIT 相似，但是其最低 4 位并不总是 0，而是以 0 表示 32 位地址，以 1 表示 64 位地址。对于 64 位地址，PCI 桥上还有 PCI_PREF_BASE_UPPER32 和 PCI_PREF_LIMIT_UPPER32 两个 32 位寄存器用于地址的高 32 位。

回到 pci_do_scan_bus()的代码中，现在一条 PCI 总线上的所有设备都已经枚举完毕了。但是，其中有些设备可能是“PCI-PCI 桥”，对这些设备还得“顺藤摸瓜”进一步扫描相应的次层 PCI 总线。对此，代码中分两趟（见 996 行）扫描当前 PCI 总线的队列 devices，如果发现一个设备的类型为“PCI-PCI 桥”或“PCI-CardBus 桥”（999 行），就对其调用 pci_scan_bridge()，这个函数的代码在 drivers/pci/pci.c 中。那么，为什么要分两趟扫描呢？这是因为在两趟扫描中针对的情况是不同的，第一趟扫描是针对已经由 BIOS 进行过处理的 PCI 桥，第二趟扫描则是针对未经 BIOS 处理的 PCI 桥。

==================== drivers/pci/pci.c 746 825 ====================

[pci_init()>pci_bios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_bridge()]

```
/*
 * If it's a bridge, configure it and scan the bus behind it.
 * For CardBus bridges, we don't scan behind as the devices will
 * be handled by the bridge driver itself.
 *
 * We need to process bridges in two passes -- first we scan those
 * already configured by the BIOS and after we are done with all of
 * them, we proceed to assigning numbers to the remaining buses in
 * order to avoid overlaps between old and new bus numbers.
 */
static int __init pci_scan_bridge(struct pci_bus *bus, struct pci_dev * dev, int max, int pass)
{
     unsigned int buses;
     unsigned short cr;
     struct pci_bus *child;
     int is_cardbus = (dev->hdr_type == PCI_HEADER_TYPE_CARDBUS);

     pci_read_config_dword(dev, PCI_PRIMARY_BUS, &buses);
     DBG("Scanning behind PCI bridge %s, config %06x, pass %d\n", dev->slot_name, buses & 0xffffff, pass);
     if ((buses & 0xffff00) && !pcibios_assign_all_busses()) {
          /*
           * Bus already configured by firmware, process it in the first
           * pass and just note the configuration.
```

```
         */
        if (pass)
            return max;
        child = pci_add_new_bus(bus, dev, 0);
        child->primary = buses & 0xFF;
        child->secondary = (buses >> 8) & 0xFF;
        child->subordinate = (buses >> 16) & 0xFF;
        child->number = child->secondary;
        if (!is_cardbus) {
            unsigned int cmax = pci_do_scan_bus(child);
            if (cmax > max) max = cmax;
        } else {
            unsigned int cmax = child->subordinate;
            if (cmax > max) max = cmax;
        }
    } else {
        /*
         * We need to assign a number to this bus which we always
         * do in the second pass. We also keep all address decoders
         * on the bridge disabled during scanning.  FIXME: Why?
         */
        if (!pass)
            return max;
        pci_read_config_word(dev, PCI_COMMAND, &cr);
        pci_write_config_word(dev, PCI_COMMAND, 0x0000);
        pci_write_config_word(dev, PCI_STATUS, 0xffff);

        child = pci_add_new_bus(bus, dev, ++max);
        buses = (buses & 0xff000000)
             | ((unsigned int)(child->primary)     <<  0)
             | ((unsigned int)(child->secondary)   <<  8)
             | ((unsigned int)(child->subordinate) << 16);
        /*
         * We need to blast all three values with a single write.
         */
        pci_write_config_dword(dev, PCI_PRIMARY_BUS, buses);
        if (!is_cardbus) {
            /* Now we can scan all subordinate buses... */
            max = pci_do_scan_bus(child);
        } else {
            /*
             * For CardBus bridges, we leave 4 bus numbers
             * as cards with a PCI-to-PCI bridge can be
             * inserted later.
             */
            max += 3;
        }
        /*
         * Set the subordinate bus number to its real value.
```

```
818              */
819             child->subordinate = max;
820             pci_write_config_byte(dev, PCI_SUBORDINATE_BUS, max);
821             pci_write_config_word(dev, PCI_COMMAND, cr);
822         }
823         sprintf(child->name, (is_cardbus ? "PCI CardBus #%02x" : "PCI Bus #%02x"), child->number);
824         return max;
825 }
```

首先从 PCI 桥的配置寄存器组中读出含有“主总线号”、“次总线号”等字段的长字，其中最低的字节为主总线号，在此之上的两个字节则依次为次总线号和子树中的最大总线号。目前，pcibios_assign_all_busses()是个空操作，固定地定义为 0，所以只要两个较高字节不为全 0 就能满足 765 行的条件。这两个字节不为全 0 说明在此之前这条总线已经枚举过一次，所以已经分配了次总线号或最大“子总线号”，一般而言这发生于系统加电时由 PCI BIOS 进行的扫描枚举。对于这样的情况，下面 772～783 行的代码只在第一趟扫描时执行。

既然是发现了一条新的 PCI 总线，当然就要为之建立一个 pci_bus 数据结构。当然，BIOS 在扫描中必定也建立了某种形式的数据结构，记录了相应的信息，但是在这里内核要撇开 BIOS，独立地建立其自己的数据结构。函数 pci_add_new_bus()的代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 712 742 ====================

[pci_init()>pci_bios_init()>pci_scan_bus()>pci_do_scan_bus()>pci_scan_bridge()>pci_add_new_bus()]

```
static struct pci_bus * __init pci_add_new_bus(struct pci_bus *parent, struct pci_dev *dev, int busnr)
{
        struct pci_bus *child;
        int i;

        /*
         * Allocate a new bus, and inherit stuff from the parent..
         */
        child = pci_alloc_bus();

        list_add_tail(&child->node, &parent->children);
        child->self = dev;
        dev->subordinate = child;
        child->parent = parent;
        child->ops = parent->ops;
        child->sysdata = parent->sysdata;

        /*
         * Set up the primary, secondary and subordinate
         * bus numbers.
         */
        child->number = child->secondary = busnr;
        child->primary = parent->secondary;
        child->subordinate = 0xff;

        /* Set up default resource pointers.. */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
738         for (i = 0; i < 4; i++)
739              child->resource[i] = &dev->resource[PCI_BRIDGE_RESOURCES+i];
740
741         return child;
742    }
```

我们把这段代码留给读者。

回到上面 pci_scan_bridge()的代码中，在进一步设置了新的 pci_bus 数据结构以后，就对这条次层 PCI 总线调用 pci_do_scan_bus()进行扫描枚举。显然，这是对 pci_do_scan_bus()的递归调用，因为 CPU 此时正是在这个函数的下面执行。可见，这是在对系统的 PCI 总线结构作深度优先的搜索。

再看当765行的条件不能满足，即首次扫描枚举一条次层PCI总线时的操作，那就是代码中的785～821 行。这一段代码仅在第二趟扫描时执行。对于未经 BIOS 处理的 PCI 桥，第一趟扫描的目的只在于知道总线号已经用到了多大，这样才可以在第二趟扫描中在此基础上继续分配总线号，并将分配的编号设置进该“ PCI-PCI 桥”的配置寄存器组。这里先读入命令寄存器的内容并保存起来，然后往命令寄存器写入全 0，往状态寄存器写入全 1（二者均为 16 位寄存器）。接着通过 pci_add_new_bus()为其建立一个 pci_bus 数据结构，注意这里调用 pci_add_new_bus()时的第三个参数 max 为次层总线的编号，这就是当前的最大总线号，并且先已加 1。然后，797 行拼组起包含着此总线号的长字，并在 804 行将其写入配置寄存器组。同样地，对于“ PCI-PCI 桥”要递归调用 pci_do_scan_bus()，这个函数返回已经在该子树中使用的最大总线号，820 行将其写入配置寄存器组中的 PCI_SUBORDINATE_BUS 字节。最后，821 行恢复命令寄存器原来的内容。

回到 pcibios_init()的代码中（987 行）。对 0 号 PCI 总线的（递归的）扫描枚举本身已经完成了，系统已经部分地（在大多数情况下实际上是全部）知道了自己的“家底”，下面就可以统筹地进行对设备的设置了。这种设置包括两方面的内容，其一是设备的中断请求线与系统中的中断控制器之间的连接，其二是设备上各个区间在总线上的地址映射。如前所述，这就是从前要通过跳线或小开关设置的两大内容。不过，实际上很可能 BIOS 已经在系统加电自检的阶段已经进行了设置，所以往往只是加以确认而已。另一方面，系统中除 0 号总线之外还可能有其它主总线存在，也需要加以扫描，但是下面读者就会看到：出于效率的考虑，这种扫描要推迟到处理中断请求线时才进行。

前面讲过，在 PCI 总线上有 INTA～INTD 共 4 条中断请求线。凡是能产生中断请求的 PCI 设备，其中断请求必定连接在其中的一条上，此时其寄存器 PCI_INTERRUPT_PIN 的内容（1～4）表示连接在哪一条线上。按 PCI 总线规格书的规定，凡是单功能 PCI 模块（接口卡）的中断请求都应该连接在 INTA 上，多功能模块（多个逻辑设备共存于同一物理模块上）才可以在使用 INTA 之余再使用其它的中断请求线。那么，这 INTA～INTD 4 条中断请求线又怎样连接到中断控制器的中断请求输入线呢？PCI 总线规格书对此并没有作硬性的规定，给具体系统的设计和实现留下了变通的余地。通常，系统的中断控制器一共有 16 条中断请求输入线，每个 PCI 设备的中断请求线（如果有的话）就是要选择连接到其中的一条上去。理想的情况当然是每条中断请求输入线最多只连接一个中断源，但是一般而言这是难以办到的，实际上只能对各项设备加以适当的组合，让若干设备共用同一条中断请求输入线。配备有 PCI 总线的 PC 机母板一般都采用一种可编程中断请求路径互连器（router，通常与 PCI-ISA 桥集成在同一芯片中），由软件设置 4 条 PCI 中断请求线与中断控制器的中断请求输入线之间的互连，图 8.4 是一种典型设计的示意图。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

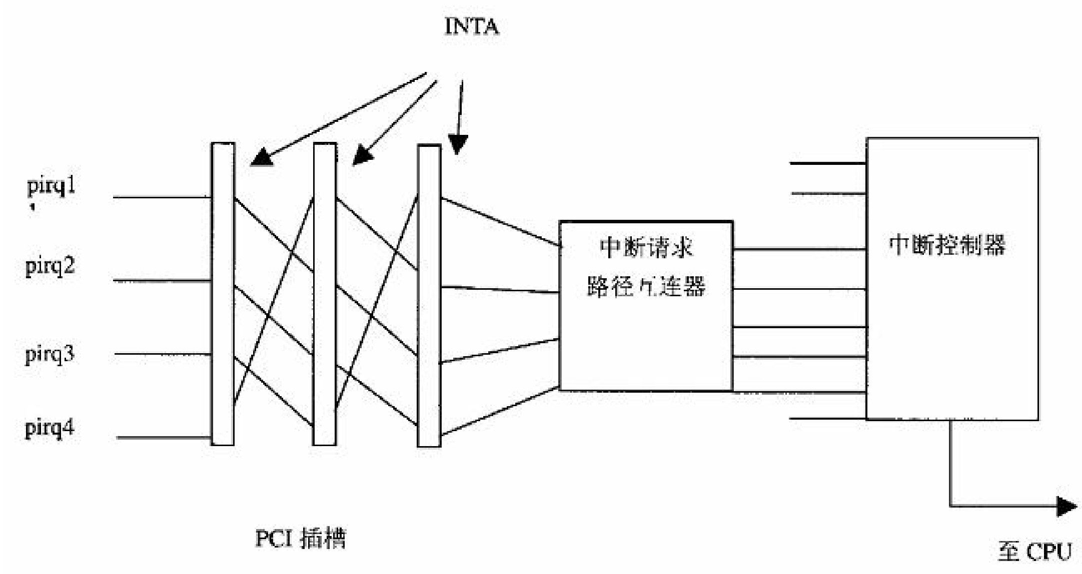


图 8.4　PCI 总线上中断请求线的互连

从图中可以看出，在 PCI 接口卡上将中断请求都连接在 INTA 上其实只有局限于具体 PCI 插槽的意义，只是使接口卡的结构比较划一而已。实际上，（这种）系统母板的设计自动将各种设备的中断请求分布到了路径互连器的各条输入线上。从系统软件的角度，我们关切的有两个方面。一是选择、确定，并通过路径互连器实施互连；二是要搞清楚具体插槽中具体设备的中断请求到底连接到了中断控制器的哪一条输入线上，这样才能确定应该把设备的中断服务程序“登记”到哪一个中断服务程序队列中（见第 3 章）。为达到这个目的，BIOS 通常提供一个“中断路径表”，为各条 PCI 总线的各个插槽提供其 4 条中断请求线的去向，就是在母板上连接到了路径互连器的哪条输入线上。至于路径互连器的输出，则总是一对一地连接到中断控制器上。

可见，对于 PCI 总线中断机制的初始化，中断路径表和路径互连器是两个关键。事实上 pcibios_irq_init()正是从中断路径表开始的，其代码在 arch/i386/kernel/pci-irq.c 中。

==================== arch/i386/kernel/pci-irq.c 516 537 ====================

[pci_init()>pcibios_init()>pcibios_irq_init()]

```
void __init pcibios_irq_init(void)
{
	DBG("PCI: IRQ init\n");
	pirq_table = pirq_find_routing_table();
#ifdef CONFIG_PCI_BIOS
	if (!pirq_table && (pci_probe & PCI_BIOS_IRQ_SCAN))
		pirq_table = pcibios_get_irq_routing_table();
#endif
	if (pirq_table) {
		pirq_peer_trick();
		pirq_find_router();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        if (pirq_table->exclusive_irqs) {
            int i;
            for (i=0; i<16; i++)
                if (!(pirq_table->exclusive_irqs & (1 << i)))
                    pirq_penalty[i] += 100;
        }
        /* If we're using the I/O APIC, avoid using the PCI IRQ routing table */
        if (io_apic_assign_pci_irqs)
            pirq_table = NULL;
    }
}
```

如上所述，PCI 总线的 4 根中断请求线与路径互连器的连接是因母板的设计而异的，所以由具体母板上的 BIOS 提供一个 PCI "中断请求路径表"，其数据结构定义于 arch/i386/kernel/pci-i386.h：

==================== arch/i386/kernel/pci-i386.h 54 65 ====================

```
struct irq_routing_table {
    u32 signature;              /* PIRQ_SIGNATURE should be here */
    u16 version;            /* PIRQ_VERSION */
    u16 size;           /* Table size in bytes */
    u8 rtr_bus, rtr_devfn;      /* Where the interrupt router lies */
    u16 exclusive_irqs;         /* IRQs devoted exclusively to PCI usage */
    u16 rtr_vendor, rtr_device; /* Vendor and device ID of interrupt router */
    u32 miniport_data;      /* Crap */
    u8 rfu[11];
    u8 checksum;            /* Modulo 256 checksum must give zero */
    struct irq_info slots[0];
} __attribute__((packed));
```

路径表的第一个长字一定是一个特殊的数值 PIRQ_SIGNATURE，实际上是 4 个特殊的字符，版本号则必须是 PIRQ_VERSION，均定义于 arch/i386/kernel/pci-i386.h 中。路径表的起点一定是与 16 字节边界对齐的，但并不固定在某个特定的位置上，所以需要扫描寻找。

==================== arch/i386/kernel/pci-irq.c 22 23 ====================

```
#define PIRQ_SIGNATURE (('$' << 0) + ('P' << 8) + ('I' << 16) + ('R' << 24))
#define PIRQ_VERSION 0x0100
```

这个 irq_routing_table 数据结构其实只是中断请求路径表的头部，表的主体 slots 是一个 irq_info 结构数组，这种数据结构的定义在 arch/i386/kernel/pci-i386.h 中：

==================== arch/i386/kernel/pci-i386.h 44 52 ====================

```
struct irq_info {
    u8 bus, devfn;              /* Bus, device and function */
    struct {
        u8 link;        /* IRQ line ID, chipset dependent, 0=not routed */
        u16 bitmap;         /* Available IRQs */
    } __attribute__((packed)) irq[4];
    u8 slot;            /* Slot number, 0=onboard */
    u8 rfu;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
} __attribute__((packed));
```

对于系统中每条 PCI 总线上的每个模块，路径表中都有个 irq_info 结构。结构中给出了其 4 条中断请求线与路径互连器输入线的连接，同时还有个位图，说明了可供选择的连接对象，即中断控制器的各条输入线。

首先通过 pirq_find_routing_table()在 BIOS 所在的区间扫描，以找到中断请求路径表，其代码在 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 46 73 ====================

[pci_init()>pcibios_init()>pcibios_irq_init()>pirq_find_routing_table()]

```
/*
 *  Search 0xf0000 -- 0xfffff for the PCI IRQ Routing Table.
 */

static struct irq_routing_table * __init pirq_find_routing_table(void)
{
     u8 *addr;
     struct irq_routing_table *rt;
     int i;
     u8 sum;

     for(addr = (u8 *) __va(0xf0000); addr < (u8 *) __va(0x100000); addr += 16) {
          rt = (struct irq_routing_table *) addr;
          if (rt->signature != PIRQ_SIGNATURE ||
              rt->version != PIRQ_VERSION ||
              rt->size % 16 ||
              rt->size < sizeof(struct irq_routing_table))
               continue;
          sum = 0;
          for(i=0; i<rt->size; i++)
               sum += addr[i];
          if (!sum) {
               DBG("PCI: Interrupt Routing Table found at 0x%p\n", rt);
               return rt;
          }
     }
     return NULL;
}
```

内核中有个全局的指针 pirq_table，指向从 BIOS 找到的中断请求路径表。如果找到了这个路径表（524 行），就要进一步加以处理了。函数 pirq_peer_trick()的代码在 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 75 110 ====================

[pci_init()>pcibios_init()>pcibios_irq_init()>pirq_peer_trick()]

```
/*
 *  If we have a IRQ routing table, use it to search for peer host
 *  bridges.  It's a gross hack, but since there are no other known
 *  ways how to get a list of buses, we have to go this way.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
79    */
80
81   static void __init pirq_peer_trick(void)
82   {
83        struct irq_routing_table *rt = pirq_table;
84        u8 busmap[256];
85        int i;
86        struct irq_info *e;
87
88        memset(busmap, 0, sizeof(busmap));
89        for(i=0; i < (rt->size - sizeof(struct irq_routing_table)) / sizeof(struct irq_info); i++) {
90             e = &rt->slots[i];
91   #ifdef DEBUG
92             {
93                  int j;
94                  DBG("%02x:%02x slot=%02x", e->bus, e->devfn/8, e->slot);
95                  for(j=0; j<4; j++)
96                       DBG(" %d:%02x/%04x", j, e->irq[j].link, e->irq[j].bitmap);
97                  DBG("\n");
98             }
99   #endif
100            busmap[e->bus] = 1;
101       }
102       for(i=1; i<256; i++)
103            /*
104             *  It might be a secondary bus, but in this case its parent is already
105             *  known (ascending bus order) and therefore pci_scan_bus returns immediately.
106             */
107            if (busmap[i] && pci_scan_bus(i, pci_root_bus->ops, NULL))
108                 printk("PCI: Discovered primary peer bus %02x [IRQ]\n", i);
109       pcibios_last_bus = -1;
110  }
```

代码中先通过 89 行的 for 循环对中断请求路径表中各个表项所涉及的总线作一番统计，循环结束以后，凡是与数组 busmap[]（以总线号为下标）中为 1 的表项相对应的总线都是在路径表中有表项的。一般而言，这些总线应该都已经在前一阶段中完成了扫描枚举，但是从 pcibios_init()中通过 pci_scan_bus()扫描的是从 0 号总线开始的一棵树，如果系统中除此之外还有其它的 PCI 主总线，则尚未受到扫描。现在，既然有了一份所有 PCI 总线的清单，就可以在其指引下有目标地通过 pci_scan_bus()扫描。这个函数对于已经扫描的 PCI 总线实际上不起作用，并返回 0（读者最好到有关的代码中验证一下）；而若返回非 0，则说明扫描到了一条未经扫描的 PCI 总线（更确切地说是一棵树），而这必定是一条主总线。最后，代码中把一个全局量 pcibios_last_bus 设成-1，表示系统中所有的主总线都已扫描，以后就不用再为此操心了。

回到 pcibios_irq_init()的代码中，补上了可能的遗漏以后，接着就要来处理路径互连器了。中断路径互连器通常与 PCI-ISA 桥集成在同一芯片中，并且也作为 PCI 设备连接在 PCI 总线上。路径表头部的 rtr_bus 和 rtr_devfn 两个字段指明了该设备所在的位置，rtr_vendor 和 rtr_device 两个字段则为芯片的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

提供者及其产品编号。函数 pirq_find_router()的作用就是找到这个设备的 pci_dev 数据结构，并根据其提供者及产品编号找到相应的驱动函数，其代码也在 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 347 355 ====================

[pci_init()>pcibios_init()>pcibios_irq_init()>pirq_find_router()]

```
347  static struct irq_router *pirq_router;
348  static struct pci_dev *pirq_router_dev;
349
350  static void __init pirq_find_router(void)
351  {
352       struct irq_routing_table *rt = pirq_table;
353       struct irq_router *r;
354
355  #ifdef CONFIG_PCI_BIOS
     . . . . . .
```

==================== arch/i386/kernel/pci-irq.c 361 387 ====================

```
361  #endif
362       /* fall back to default router if nothing else found */
363       pirq_router = pirq_routers + sizeof(pirq_routers) / sizeof(pirq_routers[0]) - 1;
364
365       pirq_router_dev = pci_find_slot(rt->rtr_bus, rt->rtr_devfn);
366       if (!pirq_router_dev) {
367            DBG("PCI: Interrupt router not found at %02x:%02x\n", rt->rtr_bus, rt->rtr_devfn);
368            return;
369       }
370
371       for(r=pirq_routers; r->vendor; r++) {
372            /* Exact match against router table entry? Use it! */
373            if (r->vendor == rt->rtr_vendor && r->device == rt->rtr_device) {
374                 pirq_router = r;
375                 break;
376            }
377            /* Match against router device entry? Use it as a fallback */
378            if (r->vendor == pirq_router_dev->vendor && r->device == pirq_router_dev->device) {
379                 pirq_router = r;
380            }
381       }
382       printk("PCI: Using IRQ router %s [%04x/%04x] at %s\n",
383            pirq_router->name,
384            pirq_router_dev->vendor,
385            pirq_router_dev->device,
386            pirq_router_dev->slot_name);
387  }
```

中断路径互连器由一个 irq_router 数据结构代表，定义于 arch/i386/kernel/pci-irq.c：

==================== arch/i386/kernel/pci-irq.c 39 44 ====================

```
39   struct irq_router {
40        char *name;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
41        u16 vendor, device;
42        int (*get)(struct pci_dev *router, struct pci_dev *dev, int pirq);
43        int (*set)(struct pci_dev *router, struct pci_dev *dev, int pirq, int new);
44    };
```

结构中的函数指针 get 和 set 用于读出和改变芯片中的互连，不同的芯片可能有不同的 get 和 set 函数。PC 机中可能采用的此类芯片有好几种，arch/i386/kernel/pci-irq.c 中定义了一个 irq_router 结构数组 pirq_routers[]，里面包括了各种常用的芯片。为减小篇幅，我们从中删去了一部分。

==================== arch/i386/kernel/pci-irq.c 323 325 ====================

```
323  static struct irq_router pirq_routers[] = {
324       { "PIIX", PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82371FB_0, pirq_piix_get, pirq_piix_set },
325       { "PIIX", PCI_VENDOR_ID_INTEL, PCI_DEVICE_ID_INTEL_82371SB_0, pirq_piix_get, pirq_piix_set },
     . . . . . .
```

==================== arch/i386/kernel/pci-irq.c 330 332 ====================

```
330       { "ALI", PCI_VENDOR_ID_AL, PCI_DEVICE_ID_AL_M1533, pirq_ali_get, pirq_ali_set },
331
332       { "VIA", PCI_VENDOR_ID_VIA, PCI_DEVICE_ID_VIA_82C586_0, pirq_via_get, pirq_via_set },
     . . . . . .
```

==================== arch/i386/kernel/pci-irq.c 336 338 ====================

```
336       { "OPTI", PCI_VENDOR_ID_OPTI, PCI_DEVICE_ID_OPTI_82C700, pirq_opti_get, pirq_opti_set },
337
338       { "NatSemi", PCI_VENDOR_ID_CYRIX, PCI_DEVICE_ID_CYRIX_5520, pirq_cyrix_get, pirq_cyrix_set },
     . . . . . .
```

==================== arch/i386/kernel/pci-irq.c 344 345 ====================

```
344       { "default", 0, 0, NULL, NULL }
345  };
```

内核中有个全局的指针 pirq_router，用来指向中断路径互连器的 irq_router 数据结构。先通过 pci_find_slot()找到其 pci_dev 数据结构。如果找不到就让指针 pirq_router 指向 pirq_routers[]中的最后一项“default”，即不存在 get 和 set 两个函数。否则就在 pirq_routers[]中找到所用的中断路径互连器芯片，并使 pirq_router 指向相应的 irq_router 数据结构。

此外，路径表中还有个位图 exclusive_irqs，位图中为 1 的位表示中断控制器的相应输入应该专用，而避免为多个中断源所共用。所以，对于这些中断请求输入线，代码中在一个数组 pirq_penalty[]的相应元素上增加了一个“惩罚量”100，使得在选择中断路径互连时不太会选择这些作为对象。

还要注意，中断路径互连器仅在采用常规的 8259A 中断控制器时才使用，所以若系统采用“高级可编程中断控制器”APIC 就在前面的 535 行把全局指针 pirq_table 设成 0，因为 APIC 本来就是“可编程”的。

那么，要是在 BIOS 中找不到中断请求路径表或者找不到路径互连器怎么办呢？那就只要采用默认的连接，不需要设置。

前面，在 pirq_peer_trick()中，根据中断路径表的指引对系统中除 0 号总线以外的主总线进行了有目标的补充扫描。但是，如果根本就没有路径表呢？那就只好进行穷举式的扫描了。当然，穷举扫描的效率比有目标的扫描要低，这就是为什么把对其它主总线的扫描推迟到找到了（或找不到）中断路径表以后的原因。当然，其代价是使代码更难懂了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

函数 pcibios_fixup_peer_bridges()对除 0 号总线以外的主总线进行穷举扫描。但是如果已经在路径表的指引下进行了扫描，则全局量 pcibios_last_bus 为负（见前面 pirq_peer_trick()的代码），因而立即就返回。这个函数的代码在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 776 805 ====================

[pci_init()>pcibios_init()>pcibios_fixup_peer_bridges()]

```
/*
 * Discover remaining PCI buses in case there are peer host bridges.
 * We use the number of last PCI bus provided by the PCI BIOS.
 */
static void __init pcibios_fixup_peer_bridges(void)
{
    int n;
    struct pci_bus bus;
    struct pci_dev dev;
    u16 l;

    if (pcibios_last_bus <= 0 || pcibios_last_bus >= 0xff)
        return;
    DBG("PCI: Peer bridge fixup\n");
    for (n=0; n <= pcibios_last_bus; n++) {
        if (pci_bus_exists(&pci_root_buses, n))
            continue;
        bus.number = n;
        bus.ops = pci_root_ops;
        dev.bus = &bus;
        for(dev.devfn=0; dev.devfn<256; dev.devfn += 8)
            if (!pci_read_config_word(&dev, PCI_VENDOR_ID, &l) &&
                l != 0x0000 && l != 0xffff) {
                DBG("Found device at %02x:%02x [%04x]\n", n, dev.devfn, l);
                printk("PCI: Discovered peer bus %02x\n", n);
                pci_scan_bus(n, pci_root_ops, NULL);
                break;
            }
    }
}
```

我们把这个函数留给读者。

回到 pcibios_init()的代码中，我们继续阅读有关中断请求路径互连的代码，函数 pcibios_fixup_irqs()的代码在 arch/i386/kernel/pci-irq.c 中， 我们抽去了采用 APIC 时的条件编译部分。

==================== arch/i386/kernel/pci-irq.c 539 562 ====================

[pci_init()>pcibios_init()>pcibios_fixup_irqs()]

```
void __init pcibios_fixup_irqs(void)
{
    struct pci_dev *dev;
    u8 pin;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    DBG("PCI: IRQ fixup\n");
    pci_for_each_dev(dev) {
        /*
         * If the BIOS has set an out of range IRQ number, just ignore it.
         * Also keep track of which IRQ's are already in use.
         */
        if (dev->irq >= 16) {
            DBG("%s: ignoring bogus IRQ %d\n", dev->slot_name, dev->irq);
            dev->irq = 0;
        }
        /* If the IRQ is already assigned to a PCI device, ignore its ISA use penalty */
        if (pirq_penalty[dev->irq] >= 100 && pirq_penalty[dev->irq] < 100000)
            pirq_penalty[dev->irq] = 0;
        pirq_penalty[dev->irq]++;
    }

    pci_for_each_dev(dev) {
        pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
#ifdef CONFIG_X86_IO_APIC
```

==================== arch/i386/kernel/pci-irq.c 595 602 ====================

```
#endif
        /*
         * Still no IRQ? Try to lookup one...
         */
        if (pin && !dev->irq)
            pcibios_lookup_irq(dev, 0);
    }
}
```

这里的 pci_for_each_dev()是个宏定义，是个对全局 pci_dev 数据结构队列 pci_devices 的循环，定义于 include/linux/pci.h：

==================== include/linux/pci.h 305 306 ====================

```
#define pci_for_each_dev(dev) \
    for(dev = pci_dev_g(pci_devices.next); dev != pci_dev_g(&pci_devices); dev = 
pci_dev_g(dev->global_list.next))
```

如前所述，每个中断请求最终都要连接到中断控制器的某条中断请求输入线上。不过，中断控制器的 16 条输入线并不是可以随意选用的，例如 0 号中断请求线就应该由时钟中断专用。另一方面，还要尽可能把各项 PCI 设备均匀地分布到不同的中断请求线上。为此，内核中设立了一个（整数）数组 pirq_penalty[]，与一个简单的算法结合在一起，就可以确定中断请求输入线的选择，这个数组定义于 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 27 37 ====================

```
/*
 * Never use: 0, 1, 2 (timer, keyboard, and cascade)
 * Avoid using: 13, 14 and 15 (FP error and IDE).
 * Penalize: 3, 4, 6, 7, 12 (known ISA uses: serial, floppy, parallel and mouse)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 */
unsigned int pcibios_irq_mask = 0xfff8;

static int pirq_penalty[16] = {
     1000000, 1000000, 1000000, 1000, 1000, 0, 1000, 1000,
     0, 0, 0, 0, 1000, 100000, 100000, 100000
};
```

数组的大小为 16，对应着 16 条中断请求输入线。每个元素代表着因选用相应中断请求输入线而受到的“惩罚”（penalty），所以选择时总是要选用受惩罚最小的。中断请求输入线 0、1、2、13、14 和 15 的惩罚一开始就是 1000000，所以实际上永远不会选用这几条输入线。一开始时，中断请求输入线 5、8、9、10 和 11 的惩罚为 0，所以开始时总是在这几条线中选择。对于同一条中断请求输入线，选用它的设备愈多，则再次选用它时受到的“惩罚”就愈大。所以，只要设备的数量不是太大，则中断请求输入线 3、4、6、7 和 12 也实际上不会被选用。而负荷的均匀分布，则在选择的过程中自然得到了保证。

代码中通过一个 for 循环（545 行）扫描系统中所有的 PCI 设备，对使用中断控制器各条输入线的设备计数（557 行），从而在 pirq_penalty[]中累积起选用各条中断请求输入线的惩罚量。中断控制器的有些输入线最好能保留给 ISA 总线上的设备专用，路径表中通过一个位图指明了这些输入线。为此，前面 pcibios_irq_init()中在 pirq_penalty[]的相应元素上增加了一个惩罚量 100。可是，那只是为了让 PCI 设备避免使用这些输入线，如果发现某个 PCI 设备已经在使用这样的输入线，那就又另当别论了。既然已经有 PCI 设备在用，戒规已经打破，那就不妨让别的 PCI 设备也来使用，所以这里（556 行）干脆把惩罚量设成 0，重新计数。

然后，又通过 for 循环再来一次对所有 PCI 设备的扫描（560 行）。如果从寄存器 PCI_INTERRUPT_PIN 读入的内容非 0，就表示该设备有中断功能。此时如果 dev->irq 为 0，即尚不知道与中断控制器的哪一条中断请求输入线相连，就要通过 pcibios_lookup_irq()寻找。寻找什么呢？如前所述，如果知道一个设备的中断请求连到了路径互连器的哪一条输入线，并且发现这条线已经在路径互连器中连接到了中断控制器，也就知道了或者说发现了该设备的中断请求的最终去向。但是，如果在路径互连器中尚未连接呢？此时是否要在中断控制器的输入线中作出选择并完成连接呢？这是由第二个调用参数决定的。这里（600 行），把参数设成 0，表示如果尚未连接就留着再说。这个函数的代码在 arch/i386/kernel/pci-irq.c 中，这个函数比较长，我们分段阅读。

==================== arch/i386/kernel/pci-irq.c 405 459 ====================
[pci_init()>pcibios_init()>pcibios_fixup_irqs()>pcibios_lookup_irq()]

```
static int pcibios_lookup_irq(struct pci_dev *dev, int assign)
{
     u8 pin;
     struct irq_info *info;
     int i, pirq, newirq;
     int irq = 0;
     u32 mask;
     struct irq_router *r = pirq_router;
     struct pci_dev *dev2;
     char *msg = NULL;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
415
416         if (!pirq_table)
417              return 0;
418
419         /* Find IRQ routing entry */
420         pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
421         if (!pin) {
422              DBG(" -> no interrupt pin\n");
423              return 0;
424         }
425         pin = pin - 1;
426
427         DBG("IRQ for %s:%d", dev->slot_name, pin);
428         info = pirq_get_info(dev);
429         if (!info) {
430              DBG(" -> not found in routing table\n");
431              return 0;
432         }
433         pirq = info->irq[pin].link;
434         mask = info->irq[pin].bitmap;
435         if (!pirq) {
436              DBG(" -> not routed\n");
437              return 0;
438         }
439         DBG(" -> PIRQ %02x, mask %04x, excl %04x", pirq, mask, pirq_table->exclusive_irqs);
440         mask &= pcibios_irq_mask;
441
442         /*
443          * Find the best IRQ to assign: use the one
444          * reported by the device if possible.
445          */
446         newirq = dev->irq;
447         if (!newirq && assign) {
448              for (i = 0; i < 16; i++) {
449                   if (!(mask & (1 << i)))
450                        continue;
451                   if (pirq_penalty[i] < pirq_penalty[newirq] &&
452                       !request_irq(i, pcibios_test_irq_handler, SA_SHIRQ, "pci-test", dev)) {
453                        free_irq(i, dev);
454                        newirq = i;
455                   }
456              }
457         }
458         DBG(" -> newirq=%d", newirq);
459
```

先从设备的配置寄存器组读入寄存器 PCI_INTERRUPT_PIN，以得知其中断请求连在 PCI 总线的哪一条中断请求线上。这个数值是从 1 开始的，所以要把它调整成从 0 开始（425 行）。然后，通过

PDF-XCHANGE Click to buy NOW! www.docu-track.com

pirq_get_info()从 BIOS 提供的中断路径表中找到所在总线和插槽的路径信息，这个函数的代码在 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 389 399 ====================

[pci_init()>pcibios_init()>pcibios_fixup_irqs()>pcibios_lookup_irq()>pirq_get_info()]

```
static struct irq_info *pirq_get_info(struct pci_dev *dev)
{
     struct irq_routing_table *rt = pirq_table;
     int entries = (rt->size - sizeof(struct irq_routing_table)) / sizeof(struct irq_info);
     struct irq_info *info;

     for (info = rt->slots; entries--; info++)
          if (info->bus == dev->bus->number && PCI_SLOT(info->devfn) == PCI_SLOT(dev->devfn))
               return info;
     return NULL;
}
```

如果根据总线号和插槽号找到了相应的 irq_info 数据结构，那么这个结构中的数组 irq[]记录着这个特定插槽的 4 条中断请求线跟中断路径互连器的连接。对于每条中断请求线，数据结构中提供了两个字段。其中 link 的主体是路径互连器的输入线号；bitmap 则是一个位图，表示哪一些中断控制器的输入是可以选择作为目标的。在 bitmap 的基础上，内核中还有个全局的屏蔽位图 pcibios_irq_mask，定义为 0xfff8，表示中断请求号 0、1、2 是不能选用的（440 行）。

我们继续往下看 pcibios_lookup_irq()的代码（arch/i386/kernel/pci-irq.c）。

==================== arch/i386/kernel/pci-irq.c 460 496 ====================

[pci_init()>pcibios_init()>pcibios_fixup_irqs()>pcibios_lookup_irq()]

```
     /* Check if it is hardcoded */
     if ((pirq & 0xf0) == 0xf0) {
          irq = pirq & 0xf;
          DBG(" -> hardcoded IRQ %d\n", irq);
          msg = "Hardcoded";
          if (dev->irq && dev->irq != irq) {
               printk("IRQ routing conflict in pirq table! Try 'pci=autoirq'\n");
               return 0;
          }
     } else if (r->get && (irq = r->get(pirq_router_dev, dev, pirq))) {
          DBG(" -> got IRQ %d\n", irq);
          msg = "Found";
          /* We refuse to override the dev->irq information. Give a warning! */
          if (dev->irq && dev->irq != irq) {
               printk("IRQ routing conflict in pirq table! Try 'pci=autoirq'\n");
               return 0;
          }
     } else if (newirq && r->set && (dev->class >> 8) != PCI_CLASS_DISPLAY_VGA) {
          DBG(" -> assigning IRQ %d", newirq);
          if (r->set(pirq_router_dev, dev, pirq, newirq)) {
               eisa_set_level_irq(newirq);
               DBG(" ... OK\n");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                msg = "Assigned";
                irq = newirq;
            }
        }

        if (!irq) {
            DBG(" ... failed\n");
            if (newirq && mask == (1 << newirq)) {
                msg = "Guessed";
                irq = newirq;
            } else
                return 0;
        }
        printk("PCI: %s IRQ %d for device %s\n", msg, irq, dev->slot_name);

```

在由路径表提供的信息中，字段 link 的高 4 位为 1 时表示路径互连器内部的连接是“硬连接”，此时其低 4 位就是中断控制器的输入线号；否则便表示路径互连器内部的连接可以通过 get 和 set 两个函数指针读出或设置。所以，对于一般的路径互连器，只要其函数指针 get 非 0，就可以通过它读出连接的目标。如果这个函数的返回值（irq）非 0，那就是所连接的目标，否则说明尚未连接。尚未连接又怎么办呢？前面讲过了，留着再说，后面读者会看到究竟留到什么时候再说。

如果已经连接，则我们已经搞清楚了目标设备中断请求的最终去向，把这个信息记录在 dev->irq 中就可以了。但是，系统中可能有很多 PCI 设备都连在同一条中断请求线上，因而有着相同的去向，应该与这些设备分享这个信息。我们继续往下看 pcibios_lookup_irq()的代码（arch/i386/kernel/pci-irq.c）。

==================== arch/i386/kernel/pci-irq.c 497 514 ====================
[pci_init()>pcibios_init()>pcibios_fixup_irqs()>pcibios_lookup_irq()]

```
        /* Update IRQ for all devices with the same pirq value */
        pci_for_each_dev(dev2) {
            pci_read_config_byte(dev2, PCI_INTERRUPT_PIN, &pin);
            if (!pin)
                continue;
            pin--;
            info = pirq_get_info(dev2);
            if (!info)
                continue;
            if (info->irq[pin].link == pirq) {
                dev2->irq = irq;
                pirq_penalty[irq]++;
                if (dev != dev2)
                    printk("PCI: The same IRQ used for device %s\n", dev2->slot_name);
            }
        }
        return 1;
    }
```

这段代码就很简单了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

搞清了中断请求的去向，下一步就是为各个 PCI 设备中的各个地址区间分配总线地址，并设置好各个 PCI 设备对这些区间（到总线地址）的映射了。我们在前面已经看到，每个 PCI 设备都通过配置寄存器组提供其各个区间的起始地址和区间大小，但那可能只是在设备内部的地址，或者是由 BIOS 分配的总线地址。所以，对于前者需要在一个统一的总线地址空间为这些区间分配地址并建立映射，对于后者则要加以验证和确认，并为之建立起相应的数据结构。对于 CPU 来说，总线地址相当于物理地址，以后在此基础上还要再加上一层映射，将虚拟地址映射到总线地址。在分配的过程中，只要原来已经分配的地址可用，就尽量维持原状，对这些地址只是验证一下，并为之建立起相应的数据结构。那么，什么样的地址才是不可用的从而需要另行分配的呢？后面读者将会看到详情，这里先简单说一下。首先，在系统初始化以后，已经把物理地址空间低端的一大块分配给了内核本身，这些地址当然不能再用于 PCI 总线，而设备内部的地址都在低端，因此需要为之另行分配地址。其次，PCI 设备的各个区间不允许互相冲突，如果发生冲突也要作出调整。对于 I/O 地址也是一样。

对于存储器和 I/O 两种地址资源的管理，内核中有一套地址资源管理机制。每一个逻辑意义上独立的连续地址区间都以一个 resource 数据结构代表，定义于 include/linux/ioport.h：

==================== include/linux/ioport.h 11 20 ====================

```
/*
 * Resources are tree-like, allowing
 * nesting etc..
 */
struct resource {
	const char *name;
	unsigned long start, end;
	unsigned long flags;
	struct resource *parent, *sibling, *child;
};
```

结构中的 start 和 end 表示该区间的地址范围，flags 是表示区间性质的一些标志位。指针 child、sibling 和 parent 则用来维系可以上下两个方向攀援的树形结构。每个区间（的 resource 结构）都通过指针 child 指向其第一个子区间，而同一区间的所有子区间则通过指针 sibling 形成一个单链，并都通过指针 parent 指向其父区间。例如，假定有一块 2KB 的地址空间 0x1000～0x17ff，这区间已经分配给某项特定的用途，假定是在一块 4 口串行接口卡上用作接收缓冲区，所以已经有了一个 resource 数据结构。然后，对于此项特定的用途，又把其中 0x1000～0x11ff、0x1200～0x13ff 和 0x1400～0x15ff 三个 512 字节的子区间进一步分配给了这块卡上的三个串行口，但是还剩下最后 512 字节尚未分配出去，那么在 2KB 的 resource 数据结构下面就有一个 3 个 resource 数据结构的队列，如图 8.5 所示。以后，当要为第 4 个串行口分配缓冲区时，先在 2KB 的 resource 数据结构中找到该区间的顶端为 0x17ff，而在其子区间队列中则找到已经分配的最高地址为 0x15ff，并且在此之前并无合适的空洞可供分配，所以正好可以把最后 512 字节分配出去，并且在其子区间队列中再挂上一个 0x1600～0x17ff 的 resource 数据结构。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

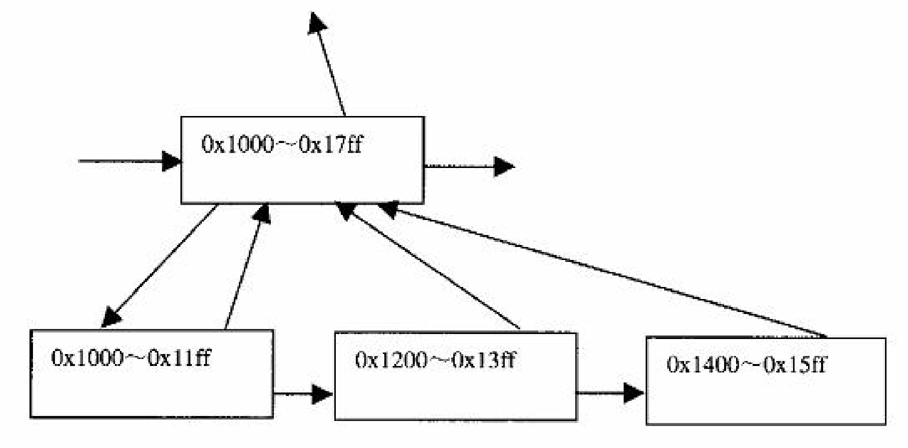


图 8.5　地址空间分配示意图

最初时，系统中只有两个区间，一个代表着 I/O 地址空间，另一个代表着内存地址。前面讲过，对于 I/O 地址空间只能通过 I/O 指令访问，而对内存地址空间则只能通过访内指令访问。这两个区间的 resource 结构定义于 kernel/resource.c：

==================== kernel/resource.c 18 19 ====================

```
struct resource ioport_resource = { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
struct resource iomem_resource = { "PCI mem", 0x00000000, 0xffffffff, IORESOURCE_MEM };
```

我们在前面也已看到，在为主 PCI 总线建立 pci_bus 结构时，它的两个区间指针一个指向 ioport_resource，表示如果需要 I/O 地址区间就从 ioport_resource 中分配；另一个指向 iomem_resource，表示如果需要内存地址区间就从 iomem_resource 中分配。不过，系统在初始化阶段已经从这两个空间中分配了许多资源，所以已经不再是像它们的初值所表示的整个 I/O 地址空间或整个内存地址空间了。

对总线地址的确认与分配是由 pcibios_resource_survey()完成的，其代码在 arch/i386/kernel/pci-i386.c 中：

==================== arch/i386/kernel/pci-i386.c 297 304 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()]

```
void __init pcibios_resource_survey(void)
{
	DBG("PCI: Allocating resources\n");
	pcibios_allocate_bus_resources(&pci_root_buses);
	pcibios_allocate_resources(0);
	pcibios_allocate_resources(1);
	pcibios_assign_resources();
}
```

首先通过 pcibios_allocate_bus_resources()为每条 PCI 总线分配地址资源，其代码在同一文件（arch/i386/kernel/pci-i386.c）中：

==================== arch/i386/kernel/pci-i386.c 152 208 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_allocate_bus_resources()]

```
/*
 * Handle resources of PCI devices.  If the world were perfect, we could
 * just allocate all the resource regions and do nothing more.  It isn't.
 * On the other hand, we cannot just re-allocate all devices, as it would
 * require us to know lots of host bridge internals.  So we attempt to
 * keep as much of the original configuration as possible, but tweak it
 * when it's found to be wrong.
 *
 * Known BIOS problems we have to work around:
 *  - I/O or memory regions not configured
 *  - regions configured, but not enabled in the command register
 *  - bogus I/O addresses above 64K used
 *  - expansion ROMs left enabled (this may sound harmless, but given
 *    the fact the PCI specs explicitly allow address decoders to be
 *    shared between expansion ROMs and other resource regions, it's
 *    at least dangerous)
 *
 * Our solution:
 *  (1) Allocate resources for all buses behind PCI-to-PCI bridges.
 *      This gives us fixed barriers on where we can allocate.
 *  (2) Allocate resources for all enabled devices.  If there is
 *      a collision, just mark the resource as unallocated. Also
 *      disable expansion ROMs during this step.
 *  (3) Try to allocate resources for disabled devices.  If the
 *      resources were assigned correctly, everything goes well,
 *      if they weren't, they won't disturb allocation of other
 *      resources.
 *  (4) Assign new addresses to resources which were either
 *      not configured at all or misconfigured.  If explicitly
 *      requested by the user, configure expansion ROM address
 *      as well.
 */

static void __init pcibios_allocate_bus_resources(struct list_head *bus_list)
{
     struct list_head *ln;
     struct pci_bus *bus;
     struct pci_dev *dev;
     int idx;
     struct resource *r, *pr;

     /* Depth-First Search on bus tree */
     for (ln=bus_list->next; ln != bus_list; ln=ln->next) {
          bus = pci_bus_b(ln);
          if ((dev = bus->self)) {
               for (idx = PCI_BRIDGE_RESOURCES; idx < PCI_NUM_RESOURCES; idx++) {
                    r = &dev->resource[idx];
                    if (!r->start)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
200                         continue;
201                    pr = pci_find_parent_resource(dev, r);
202                    if (!pr || request_resource(pr, r) < 0)
203                         printk(KERN_ERR "PCI: Cannot allocate resource region %d of bridge %s\n", idx,
dev->slot_name);
204               }
205          }
206          pcibios_allocate_bus_resources(&bus->children);
207     }
208  }
```

参数 bus_list 指向一个队列头，实际上是一个 pci_bus 结构队列，第一次调用这个函数时指向队列 pci_root_buses，即所有通过“宿主—PCI 桥”相连的 PCI 总线（通常只有一条）。如前所述，每条 PCI 总线都有可能通过“PCI-PCI 桥”连接到更次层的其它 PCI 总线，所以每个 pci_bus 数据结构中都有个队列头 children，用来维持次层 PCI 总线的 pci_bus 结构队列。完成了对 PCI 总线和设备的枚举以后，这些数据结构都已经各就各位了。既然 PCI 总线的系统结构是递归的，对整个 PCI 结构的资源分配就自然也是递归的，所以 206 行对次层 pci_bus 结构队列递归调用 pcibios_allocatebus_resources()，向下作深度优先的遍历。代码中外层的 for 循环（194 行）是对同一队列中所有 pci_bus 结构的循环。对于总线本身，即 PCI 桥，由内层的 for 循环（197 行）对其 4 个（从 7 至 10）地址区间加以检验。

这里的常数 PCI_BRIDGE_RESOURCES 和 PCI_NUM_RESOURCES 均定义于 include/linux/pci.h：

==================== include/linux/pci.h 267 377 ====================

```
/*
 * For PCI devices, the region numbers are assigned this way:
 *
 *   0-5  standard PCI regions
 *   6    expansion ROM
 *   7-10 bridges: address space assigned to buses behind the bridge
 */

#define PCI_ROM_RESOURCE 6
#define PCI_BRIDGE_RESOURCES 7
#define PCI_NUM_RESOURCES 11
```

对于普通的 PCI 设备，其 pci_dev 结构中的开头 6 个（0 至 5）地址区间是设备上可能有的区间，第 7 个区间（6）是可能有的扩充 ROM 区间。如果这设备是个 PCI 桥，则后面还有 4 个区间，pci_bus 结构中的 4 个 resource 指针就分别指向这 4 个区间。前面，我们在阅读 pci_read_bridge_bases()时说过，PCI 桥本身并不“使用”这些区间中的地址，而是用这些区间作为地址过滤的窗口。其中第一个窗口用于 I/O 地址，第二个用于存储器地址，第三个则为“可预取”存储器地址区间。此外，还有一个用于扩充 ROM 区间的窗口。次层总线上所有设备（包括 PCI 桥）所使用的地址都必须在这些窗口中，换言之，这些设备所需要的地址资源都要从这些区间中分配。所以，每个 PCI 桥或者说每条 PCI 总线，都需要从其上层“批发”下一些地址资源，然后“零售”分配给连接在这条总线上的所有设备，包括把其中的一部分批发给更次层总线。就这样，每一条 PCI 总线上的设备都向其所在的总线批发地址资源，而总线则向其上层总线批发。那么，顶层的 PCI 总线又向谁批发呢？那就是 ioport_resource 和

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

iomem_resource，这是两种地址资源终极的来源。

如果 PCI 桥的某个区间已经有了对资源的需求，就要先通过 pci_find_parent_resource()看看其“父节点”是否拥有其所需的地址资源，其代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 164 194 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_allocate_bus_resources()>pci_find_parent_resource()]

```
164 /**
165  * pci_find_parent_resource - return resource region of parent bus of given region
166  * @dev: PCI device structure contains resources to be searched
167  * @res: child resource record for which parent is sought
168  *
169  *  For given resource region of given device, return the resource
170  *  region of parent bus the given region is contained in or where
171  *  it should be allocated from.
172  */
173 struct resource *
174 pci_find_parent_resource(const struct pci_dev *dev, struct resource *res)
175 {
176      const struct pci_bus *bus = dev->bus;
177      int i;
178      struct resource *best = NULL;
179 
180      for(i=0; i<4; i++) {
181           struct resource *r = bus->resource[i];
182           if (!r)
183                continue;
184           if (res->start && !(res->start >= r->start && res->end <= r->end))
185                continue;/* Not contained */
186           if ((res->flags ^ r->flags) & (IORESOURCE_IO | IORESOURCE_MEM))
187                continue;/* Wrong type */
188           if (!((res->flags ^ r->flags) & IORESOURCE_PREFETCH))
189                return r;/* Exact match */
190           if ((res->flags & IORESOURCE_PREFETCH) && !(r->flags & IORESOURCE_PREFETCH))
191                best = r;/* Approximating prefetchable by non-prefetchable */
192      }
193      return best;
194 }
```

参数 dev 指向代表着次层总线的 PCI 桥的 pci_dev 数据结构，res 则指向代表着所需地址区间的 resource 结构。分配时依次扫描 PCI 桥所在总线的 4 个地址区间。

分配的大原则是地址的范围必须相符（185 行），并且类型（存储器地址或 I/O 地址）必须相符（187 行）。其次，是否“可预取”也最好能一致。要不然，如果所要求的区间用于“可预取”的存储器，而总线上的区间本来是供不可预取的寄存器使用的，那么虽然有些勉强，也还不失为一个选择（191 和 193 行，反过来就不行了）。

如果在父节点中找到了能够满足要求的区间，则函数返回指向该区间的指针（否则为 0），说明所需的地址资源是有保障的，所以就通过 request_resource()加以分配，这个函数的代码在 kernel/resource.c

中：

==================== kernel/resource.c 114 122 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_allocate_bus_resources()>request_resource()]

```
int request_resource(struct resource *root, struct resource *new)
{
	struct resource *conflict;

	write_lock(&resource_lock);
	conflict = __request_resource(root, new);
	write_unlock(&resource_lock);
	return conflict ? -EBUSY : 0;
}
```

落实资源分配的过程涉及队列操作，不容许受到打扰，所以必须加锁。具体的操作则由__request_resource()完成，其代码也在同一文件中：

==================== kernel/resource.c 66 93 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_allocate_bus_resources()>request_resource()>__request_resource()]

```
/* Return the conflict entry if you can't request it */
static struct resource * __request_resource(struct resource *root, struct resource *new)
{
	unsigned long start = new->start;
	unsigned long end = new->end;
	struct resource *tmp, **p;

	if (end < start)
		return root;
	if (start < root->start)
		return root;
	if (end > root->end)
		return root;
	p = &root->child;
	for (;;) {
		tmp = *p;
		if (!tmp || tmp->start > end) {
			new->sibling = tmp;
			*p = new;
			new->parent = root;
			return NULL;
		}
		p = &tmp->sibling;
		if (tmp->end < start)
			continue;
		return tmp;
	}
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的参数 root 指向总线的某个 resource 数据结构，而 new 则指向 PCI 桥，即次层总线的 resource 数据结构。代码中通过一个 for 循环在 root 的子区间队列中为 new 找到适当的位置，然后将 new 插入该队列中。如果发现 new 与已经存在的子区间冲突，则操作失败而返回与其冲突的区间指针。

当完成了对 pcibios_allocate_bus_resources()的递归调用时，所有 PCI 总线所需的地址资源都已经分配好了。但是，读者大概也看出来了，除对于冲突的检验以外，这里所谓“分配”其实只是一种“事后追认”而已。这些区间的范围本来就是从 PCI 桥中读出来的，现存也并未加以改变。所做的只是为这些区间建立起 resource 结构，并插入到整个地址资源树中的某个位置上，以供检验冲突之用。那么，这些 PCI 桥是怎么知道应该有什么样的窗口的呢？这是由 BIOS 设置的。既然已经设置，也就没有必要推倒重来。如果 BIOS 并未设置，那也不难通过扫描已经建立起的 pci_bus 和 pci_dev 数据结构统计出来，或者在扫描、枚举的过程中计算出来。

回到 pcibios_resource_survey()的代码中，接着就可以为所有的 PCI 设备分配地址资源了，这里分两趟调用 pcibios_allocate_resources()，这个函数的代码在 arch/i386/kernel/pci-i386.c 中。同样，所谓“分配”实际上往往只是追认。

==================== arch/i386/kernel/pci-i386.c 210 253 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_allocate_resources()]

```
static void __init pcibios_allocate_resources(int pass)
{
	struct pci_dev *dev;
	int idx, disabled;
	u16 command;
	struct resource *r, *pr;

	pci_for_each_dev(dev) {
		pci_read_config_word(dev, PCI_COMMAND, &command);
		for(idx = 0; idx < 6; idx++) {
			r = &dev->resource[idx];
			if (r->parent)		/* Already allocated */
				continue;
			if (!r->start)		/* Address not assigned at all */
				continue;
			if (r->flags & IORESOURCE_IO)
				disabled = !(command & PCI_COMMAND_IO);
			else
				disabled = !(command & PCI_COMMAND_MEMORY);
			if (pass == disabled) {
				DBG("PCI: Resource %08lx-%08lx (f=%lx, d=%d, p=%d)\n",
				    r->start, r->end, r->flags, disabled, pass);
				pr = pci_find_parent_resource(dev, r);
				if (!pr || request_resource(pr, r) < 0) {
					printk(KERN_ERR "PCI: Cannot allocate resource region %d of device %s\n", idx,
dev->slot_name);
					/* We'll assign a new address later */
					r->end -= r->start;
					r->start = 0;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
238                  }
239             }
240         }
241         if (!pass) {
242             r = &dev->resource[PCI_ROM_RESOURCE];
243             if (r->flags & PCI_ROM_ADDRESS_ENABLE) {
244                 /* Turn the ROM off, leave the resource region, but keep it unregistered. */
245                 u32 reg;
246                 DBG("PCI: Switching off ROM of %s\n", dev->slot_name);
247                 r->flags &= ~PCI_ROM_ADDRESS_ENABLE;
248                 pci_read_config_dword(dev, dev->rom_base_reg, &reg);
249                 pci_write_config_dword(dev, dev->rom_base_reg, reg & ~PCI_ROM_ADDRESS_ENABLE);
250             }
251         }
252     }
253 }
```

这是一个对所有 pci_dev 数据结构的循环。对于每一项 PCI 设备的 6 个常规区间，如果这些区间已经生效（可以接受访问）就在第一趟扫描（pass 为 0）时分配地址资源，否则就在第一趟扫描（pass 为 1）时分配。对扩充 ROM 区间则只在第一趟，即参数 pass 为 0 时加以处理。

PCI 设备的这些区间有几种可能。第一种是已经分配了地址资源，因而其指针 parent 已指向其父节点，对这种节点无需作任何处理，所以把它跳过就行了（222 行）。第二种是区间的起始地址为 0，这种区间或者是本来就不需要为之分配空间，或者是已知其父节点暂时不能满足其需要，这里也把它跳过（224 行）。第三种就是需要分配地址资源的了，如果通过 pci_find_parent_resource()发现这部分地址资源已终包含在父节点的资源中，那就可以通过 request_resource()分配了，这两个函数的代码我们都已在前面读过。如果发现不能在当前的起始地址上从父节点分配到所需的地址资源，即地址范围不符或者发生了冲突，就先将该区间平移到起始地址为 0 的地方（236～237 行），这些区间的起始地址需要加以变更才行。这里要说明，对这些区间的地址资源分配之所以失败并不是由于父节点中的资源短缺，而是因为对起始地址的要求不能满足。BIOS 在确定父节点的窗口大小时是经过计算的，加上对窗口位置的对齐，父节点的窗口一般都要比实际的需要大。所以，只要允许将区间适当平移，就一定能分配到所需的地址资源。

对于 ROM 区间，则在第一趟扫描时予以关闭。ROM 区间一般只是在初始化时由 BIOS 或具体的设备驱动程序使用，所以现在可以关闭了。但是其地址资源暂时还保存着，如果需要还可以在设备驱动程序中再打开。

对于不能在原有起始地址上分配到所需地址资源的区间，即起始地址已改成 0 的区间，要通过 pcibios_assign_resources()加以分配，实际上这才是真正意义上的“分配”而不是追认。这个函数的代码在 arch/i386/kernel/pci-i386.c 中：

==================== arch/i386/kernel/pci-i386.c 255 295 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign resources()]

```
255 static void __init pcibios_assign_resources(void)
256 {
257     struct pci_dev *dev;
258     int idx;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct resource *r;

    pci_for_each_dev(dev) {
        int class = dev->class >> 8;

        /* Don't touch classless devices and host bridges */
        if (!class || class == PCI_CLASS_BRIDGE_HOST)
            continue;

        for(idx=0; idx<6; idx++) {
            r = &dev->resource[idx];

            /*
             *  Don't touch IDE controllers and I/O ports of video cards!
             */
            if ((class == PCI_CLASS_STORAGE_IDE && idx < 4) ||
               (class == PCI_CLASS_DISPLAY_VGA && (r->flags & IORESOURCE_IO)))
                continue;

            /*
             *  We shall assign a new address to this resource, either because
             *  the BIOS forgot to do so or because we have decided the old
             *  address was unusable for some reason.
             */
            if (!r->start && r->end)
                pci_assign_resource(dev, idx);
        }

        if (pci_probe & PCI_ASSIGN_ROMS) {
            r = &dev->resource[PCI_ROM_RESOURCE];
            r->end -= r->start;
            r->start = 0;
            if (r->end)
                pci_assign_resource(dev, PCI_ROM_RESOURCE);
        }
    }
}
```

这个函数就是个嵌套的 for 循环，对于系统中的所有 PCI 设备，只要这个设备有效（类型字段非 0），并且不是 PCI 桥，就在内层循环中检查其 6 个可能的地址区间。只要是需要分配地址资源的地址区间（起始地址为 0 而终点地址非 0），便通过 pci_assign_resource()为其分配总线地址，并将其设置入具体设备的配置寄存器组。回顾一下前面 pcibios_allocate_resources()的代码，就可以看出这些区间正是在那里没有能解决的区间。当时之所以没有能解决是因为指定了起始地址，现在则放宽了条件。

不过，IDE 存储设备（硬盘）的前 4 个区间和 VGA 显示设备的 I/O 地址区间是特例，这些区间的地址不需要分配、也不能改变（因为已经在用了）。

函数 pci_assign_resource()的代码在 drivers/pci/setup-res.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/pci/setup-res.c 99 127 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()]

```
99   int
100  pci_assign_resource(struct pci_dev *dev, int i)
101  {
102      const struct pci_bus *bus = dev->bus;
103      struct resource *res = dev->resource + i;
104      unsigned long size, min;
105
106      size = res->end - res->start + 1;
107      min = (res->flags & IORESOURCE_IO) ? PCIBIOS_MIN_IO : PCIBIOS_MIN_MEM;
108
109      /* First, try exact prefetching match.. */
110      if (pci_assign_bus_resource(bus, dev, res, size, min, IORESOURCE_PREFETCH, i) < 0) {
111          /*
112           * That failed.
113           *
114           * But a prefetching area can handle a non-prefetching
115           * window (it will just not perform as well).
116           */
117          if (!(res->flags & IORESOURCE_PREFETCH) || pci_assign_bus_resource(bus, dev, res, size, min, 0, i) < 0) {
118              printk(KERN_ERR "PCI: Failed to allocate resource %d for %s\n", i, dev->name);
119              return -EBUSY;
120          }
121      }
122
123      DBGC((" got res[%lx:%lx] for resource %d of %s\n", res->start,
124                               res->end, i, dev->name));
125
126      return 0;
127  }
```

这个函数通过 pci_assign_bus_resource()为给定设备从其所在的总线分配地址区间。为设备上的每个区间分配地址时，一方面要考虑区间的实际大小，另一方面对其位置也有所限制，对于 I/O 地址区间其起点不得低于 PCIBIOS_MIN_IO，对于内存地址区间则不得低于 PCIBIOS_MIN_MEM，这两个常数均定义于 include/asm-i386/pci.h：

==================== include/asm-i386/pci.h 12 13 ====================

```
#define PCIBIOS_MIN_IO       0x1000
#define PCIBIOS_MIN_MEM      0x10000000
```

就是说，I/O 地址区间的位置不得低于 4KB，而内存地址区间的位置不得低于 256MB。另一方面，对于地址区间还有个是否必须满足“可预取”要求的问题，所以这里（110 行）先以 IORESOURCE_PREFETCH 为参数调用一次 pci_assign_bus_resource()，表示要求在“可预取”方面相符。若不能满足要求，则根据情况再以 0 为参数调用一次（117 行），表示在这方面不相符也可以，但这只适用于设备上的区间为“可预取”，即用于存储器的情况下，反过来就不可以了。函数

pci_assign_bus_resource()的代码也在 drivers/pci/setup-res.c 中：

==================== drivers/pci/setup-res.c 59 97 ====================
[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()>pci_assign_bus_resource()]

```
/*
 * Given the PCI bus a device resides on, try to
 * find an acceptable resource allocation for a
 * specific device resource..
 */
static int pci_assign_bus_resource(const struct pci_bus *bus,
	struct pci_dev *dev,
	struct resource *res,
	unsigned long size,
	unsigned long min,
	unsigned int type_mask,
	int resno)
{
	int i;

	type_mask |= IORESOURCE_IO | IORESOURCE_MEM;
	for (i = 0 ; i < 4; i++) {
		struct resource *r = bus->resource[i];
		if (!r)
			continue;

		/* type_mask must match */
		if ((res->flags ^ r->flags) & type_mask)
			continue;

		/* We cannot allocate a non-prefetching resource from a pre-fetching area */
		if ((r->flags & IORESOURCE_PREFETCH) && !(res->flags & IORESOURCE_PREFETCH))
			continue;

		/* Ok, try it out.. */
		if (allocate_resource(r, res, size, min, -1, size, pcibios_align_resource, dev) < 0)
			continue;

		/* Update PCI config space.  */
		pcibios_update_resource(dev, r, res, resno);
		return 0;
	}
	return -EBUSY;
}
```

从哪里分配地址区间呢？设备在哪一条 PCI 总线上，就从已经成块“批发”给这条总线的地址区间中分配。这里的参数 bus 指向总线的 pci_bus 结构，而 res 则指向一个 resource 数据结构，结构中记载着对目标区间的要求。代码中通过一个 for 循环依次尝试该总线的 4 个可能的区间。如果二者类型相

PDF-XCHANGE Click to buy NOW! www.docu-track.com

符，即同为 I/O 地址或同为内存地址，并且是否“可预取”也相符（如果要求的话），就通过 allocate_resource()为其分配总线地址（kernel/resource.c），如果分配成功就通过 pcibios_update_resource()将其设置进目标设备。

==================== kernel/resource.c 185 203 ====================
[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()>pci_assign_bus_resource()>allocate_resource()]

```
/*
 * Allocate empty slot in the resource tree given range and alignment.
 */
int allocate_resource(struct resource *root, struct resource *new,
                      unsigned long size,
                      unsigned long min, unsigned long max,
                      unsigned long align,
                      void (*alignf)(void *, struct resource *, unsigned long),
                      void *alignf_data)
{
     int err;

     write_lock(&resource_lock);
     err = find_resource(root, new, size, min, max, align, alignf, alignf_data);
     if (err >= 0 && __request_resource(root, new))
          err = -EBUSY;
     write_unlock(&resource_lock);
     return err;
}
```

参数 root 指向一个 resource 数据结构，代表着设备所在总线上的一个地址区间；而 new 则指向另一个 resource 数据结构，代表着待分配的地址区间。这块地址的大小由参数 size 确定，实际上一定是 2 的某次幂，位置须在 min 与 max 之间，align 则与边界对齐有关。对照前面对这个函数的调用，就可以看出这里 size 是实际的区间大小，min 为 PCIBIOS_MIN_IO 或 PCIBIOS_MIN_MEM，max 为 0xffffffff，而 align 与 size 相同。还有，参数 alignf 是个函数指针，实际上指向 pcibios_align_resource()，alignf_data 则实际上指向目标设备的 pci_dev 数据结构。注意这里的 max 是个无符号整数，所以-1 实际上是全 1。这些参数原封不动地传给 find_resource()，先通过这个函数找到符合要求的区间，其代码在 kernel/resource.c 中：

==================== kernel/resource.c 148 183 ====================
[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()>pci_assign_bus_resource()>allocate_resource()>find_resource()]

```
/*
 * Find empty slot in the resource tree given range and alignment.
 */
static int find_resource(struct resource *root, struct resource *new,
                       unsigned long size,
                       unsigned long min, unsigned long max,
                       unsigned long align,
                       void (*alignf)(void *, struct resource *, unsigned long),
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    void *alignf_data)
{
    struct resource *this = root->child;

    new->start = root->start;
    for(;;) {
        if (this)
            new->end = this->start;
        else
            new->end = root->end;
        if (new->start < min)
            new->start = min;
        if (new->end > max)
            new->end = max;
        new->start = (new->start + align - 1) & ~(align - 1);
        if (alignf)
            alignf(alignf_data, new, size);
        if (new->start < new->end && new->end - new->start + 1 >= size) {
            new->end = new->start + size - 1;
            return 0;
        }
        if (!this)
            break;
        new->start = this->end + 1;
        this = this->sibling;
    }
    return -EBUSY;
}
```

具体的分配并不复杂。如果总线的地址区间是个整体，由 root 所指的 resource 数据结构是个叶节点，那么其起点就用作待分配区间的起点，然后将起点按要求进行边界对齐，并作必要的调整，再来检查区间的大小。否则，要是 root->child 非 0，则需要扫描各个子区间，从中发现足够大的子区间。分配成功时函数返回 0，否则返回-EBUSY。初步确定了分配的起始地址，并且作了边界对齐以后，可能（如果 alignf 非 0）还要调整一下起点地址。在这里，用来调整的函数是 pcibios_align_resource()，其代码在 arch/i386/kernel/pci-i386.c 中：

==================== arch/i386/kernel/pci-i386.c 125 149 ====================

[pci_init()>pcibios_init(_)>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()>pci_assign_bus_resource()>allocate_resource()>find_resource()>pcibios_align_resource()]

```
/*
 * We need to avoid collisions with `mirrored' VGA ports
 * and other strange ISA hardware, so we always want the
 * addresses to be allocated in the 0x000-0x0ff region
 * modulo 0x400.
 *
 * Why? Because some silly external IO cards only decode
 * the low 10 bits of the IO address. The 0x00-0xff region
 * is reserved for motherboard devices that decode all 16
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
134   * bits, so it's ok to allocate at, say, 0x2800-0x28ff,
135   * but we want to try to avoid allocating at 0x2900-0x2bff
136   * which might have be mirrored at 0x0100-0x03ff..
137   */
138  void
139  pcibios_align_resource(void *data, struct resource *res, unsigned long size)
140  {
141      if (res->flags & IORESOURCE_IO) {
142          unsigned long start = res->start;
143
144          if (start & 0x300) {
145              start = (start + 0x3ff) & ~0x3ff;
146              res->start = start;
147          }
148      }
149  }
```

这种调整只是针对 I/O 地址的（141 行）。为什么要作调整呢？这是因为在早期的 PC 机中规定外设接口卡只能使用 4KB 以下的 I/O 地址，而其中 0x00～0xff 又是保留给母板使用的。所以，早期的接口卡往往只对 I/O 地址的低 10 位，即 4KB 的范围解码，但是同时这些接口卡又不会使用 0x00～0xff，即 bit8 和 bit9 为 0 的这段区间。因此，要避免使用地址中 bit8 或 bit9 为非 0 的那些地址，以免与这样的接口卡冲突。不过，实际上 PCI 设备通常都将寄存器映射到存储器空间（而不是 I/O 空间），所以真正需要作出这种调整的机会是不多的。

回到 allocate_resource()的代码中，如果对 find_resource()的调用成功了，找到了所需的地址资源，就要进一步通过__request_resource()将代表着新分配区间的 resource 结构 new 插入 root 的 child 队列中。我们已在前面看过这个函数的代码。至此为止，这种“分配”还只是“账面”上的，只是在 resource 结构树中进行，还没有把分配到的地址设置到具体的设备中去。函数成功时返回 0，失败则返回一个非 0 指针。

为目标设备上的目标区间分配了总线地址以后，便返回到 pci_assign_bus_resource()中，接着便通过 pcibios_update_resource()将起始地址设置到目标设备中，从而建立起地址映射。这个函数的代码在 arch/i386/kernel/pci-i386.c 中：

==================== arch/i386/kernel/pci-i386.c 97 123 ====================

[pci_init()>pcibios_init()>pcibios_resource_survey()>pcibios_assign_resources()>pci_assign_resource()>pci_assign_bus_resource()>pcibios_update_resource()]

```
void
pcibios_update_resource(struct pci_dev *dev, struct resource *root,
                        struct resource *res, int resource)
{
    u32 new, check;
    int reg;

    new = res->start | (res->flags & PCI_REGION_FLAG_MASK);
    if (resource < 6) {
        reg = PCI_BASE_ADDRESS_0 + 4*resource;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
107        } else if (resource == PCI_ROM_RESOURCE) {
108            res->flags |= PCI_ROM_ADDRESS_ENABLE;
109            new |= PCI_ROM_ADDRESS_ENABLE;
110            reg = dev->rom_base_reg;
111        } else {
112            /* Somebody might have asked allocation of a non-standard resource */
113            return;
114        }
115
116        pci_write_config_dword(dev, reg, new);
117        pci_read_config_dword(dev, reg, &check);
118        if ((new ^ check) & ((new & PCI_BASE_ADDRESS_SPACE_IO) ? PCI_BASE_ADDRESS_IO_MASK : PCI_BASE_ADDRESS_MEM_MASK)) {
119            printk(KERN_ERR "PCI: Error while updating region "
120                 "%s/%d (%08x != %08x)\n", dev->slot_name, resource,
121                 new, check);
122        }
123    }
```

这里的参数 res 指向目标设备上目标区间的 resource 数据结构，其中的字段 start 就是为其分配的（起始）总线地址，整数（下标）resource 则表明是设备中的哪一个区间。如前所述，这个总线地址一定是与 16 字节边界对齐的，所以其最低 4 位用于控制目的，这些标志位来自 res->flags 的最低 4 位。总线地址的设置和映射的建立倒是很简单的，只要把所分配的总线地址连同标志位写入目标设备的配置寄存器组中相应区间的寄存器就行了。这里 116 行将新的地址通过配置寄存器组写入目标设备，然后再读回来加以验证。

不过，设置了区间的地址并不意味着这个区间已经可以访问了。配置寄存器组中的命令寄存器里有两个控制位，即 PCI_COMMAND_IO 和 PCI_COMMAND_MEMORY，就好像是两个总开关，分别控制着设备的所有 I/O 地址区间和存储器地址区间。最终，还要把这两个控制位都设成 1 才能使这些区间真正地连接到 PCI 总线上。我们看到，这里并没有走出这最后一步，实际上是留给了具体的设备驱动程序。

完成了对所有设备、所有区间的地址设置以后，逐层返回到 pci_init()中，还要在一个循环中对每个设备调用 pci_fixup_device()。我们已在前面看过它的代码，不过上一次调用时的参数 pass 为 PCI_FIXUP_HEADER，进行的是对从设备读出的头部信息的修正；而这一次为 PCI_FIXUP_FINAL，是对设置了地址以后的修正。

至此，对 PCI 总线的初始化已经完成，内存中已经建立起代表着全部 PCI 总线和 PCI 设备的若干棵（通常只是一棵）树，而每项设备的每个地址区间都已有了总线地址。这样，给定一项设备（更确切地说是一项功能）的有关信息，例如设备的类型、由谁制造，就可以通过搜索这些树找到代表着此项设备的 pci_dev 数据结构。为此，内核提供了一个函数 pci_find_device()，其代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 96 100 ====================

```
struct pci_dev *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
pci_find_device(unsigned int vendor, unsigned int device, const struct pci_dev *from)
{
	return pci_find_subsys(vendor, device, PCI_ANY_ID, PCI_ANY_ID, from);
}
```

这个函数的主体是 pci_find_subsys()，其代码也在同一文件中：

==================== drivers/pci/pci.c 63 80 ====================

```
struct pci_dev *
pci_find_subsys(unsigned int vendor, unsigned int device,
		unsigned int ss_vendor, unsigned int ss_device,
		const struct pci_dev *from)
{
	struct list_head *n = from ? from->global_list.next : pci_devices.next;

	while (n != &pci_devices) {
		struct pci_dev *dev = pci_dev_g(n);
		if ((vendor == PCI_ANY_ID || dev->vendor == vendor) &&
		    (device == PCI_ANY_ID || dev->device == device) &&
		    (ss_vendor == PCI_ANY_ID || dev->subsystem_vendor == ss_vendor) &&
		    (ss_device == PCI_ANY_ID || dev->subsystem_device == ss_device))
			return dev;
		n = n->next;
	}
	return NULL;
}
```

当 PCI 总线上有多个来自同一厂商的同种设备时（例如两块 Ethernet 接口卡），可以逐次以不同的起点 from 调用 pci_find_device()，直至找到所有这些设备的 pci_dev 数据结构。

如果关心的不是由谁制造，而是模块的功能和用途，则可以通过 pci_find_class()寻找，其代码也在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 115 127 ====================

```
struct pci_dev *
pci_find_class(unsigned int class, const struct pci_dev *from)
{
	struct list_head *n = from ? from->global_list.next : pci_devices.next;

	while (n != &pci_devices) {
		struct pci_dev *dev = pci_dev_g(n);
		if (dev->class == class)
			return dev;
		n = n->next;
	}
	return NULL;
}
```

找到了目标设备的 pci_dev 数据结构，就可以通过其指针数组 resource[]找到各个总线地址区间的 resource 结构，从而取得各个区间的总线地址。在这个基础上，通过__ioremap()为这些区间建立起虚拟

PDF-XCHANGE Click to buy NOW! www.docu-track.com

地址的映射，再通过设备的命令寄存器打开其所有区间，就可以像访问内存空间一样地访问这些区间了。此外，如果目标设备具有中断功能，则还要进一步落实好中断请求线。内核为设备驱动程序提供了一个 inline 函数 pcibios_enable_device()来帮助驱动程序做这些事情，其代码在 arch/i386/kernel/pci-pc.c 中：

==================== arch/i386/kernel/pci-pc.c 1042 1050 ====================

```
int pcibios_enable_device(struct pci_dev *dev)
{
	int err;

	if ((err = pcibios_enable_resources(dev)) < 0)
		return err;
	pcibios_enable_irq(dev);
	return 0;
}
```

首先就是通过 pcibios_enable_resources()打开设备的各个地址区间，其代码在 arch/i386/kernel/pci-i386.c 中：

==================== arch/i386/kernel/pci-i386.c 306 332 ====================

[pcibios_enable_device()>pcibios_enable_resources()]

```
int pcibios_enable_resources(struct pci_dev *dev)
{
	u16 cmd, old_cmd;
	int idx;
	struct resource *r;

	pci_read_config_word(dev, PCI_COMMAND, &cmd);
	old_cmd = cmd;
	for(idx=0; idx<6; idx++) {
		r = &dev->resource[idx];
		if (!r->start && r->end) {
			printk(KERN_ERR "PCI: Device %s not available because of resource collisions\n", dev->slot_name);
			return -EINVAL;
		}
		if (r->flags & IORESOURCE_IO)
			cmd |= PCI_COMMAND_IO;
		if (r->flags & IORESOURCE_MEM)
			cmd |= PCI_COMMAND_MEMORY;
	}
	if (dev->resource[PCI_ROM_RESOURCE].start)
		cmd |= PCI_COMMAND_MEMORY;
	if (cmd != old_cmd) {
		printk("PCI: Enabling device %s (%04x -> %04x)\n", dev->slot_name, old_cmd, cmd);
		pci_write_config_word(dev, PCI_COMMAND, cmd);
	}
	return 0;
```

```
}
```

这段代码很简单，我们就不多加解释了。

接着是对中断请求线的处理，目的是要搞清楚该设备的中断请求最终连接到了中断控制器的哪一条输入线，这样才能正确地登记其中断服务程序。函数 pcibios_enable_irq() 的代码在 arch/i386/kernel/pci-irq.c 中：

==================== arch/i386/kernel/pci-irq.c 613 628 ====================

[pcibios_enable_device()>pcibios_enable_resources()]

```
613  void pcibios_enable_irq(struct pci_dev *dev)
614  {
615       u8 pin;
616       pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
617       if (pin && !pcibios_lookup_irq(dev, 1) && !dev->irq) {
618            char *msg;
619            if (io_apic_assign_pci_irqs)
620                 msg = " Probably buggy MP table.";
621            else if (pci_probe & PCI_BIOS_IRQ_SCAN)
622                 msg = "";
623            else
624                 msg = " Please try using pci=biosirq.";
625            printk(KERN_WARNING "PCI: No IRQ known for interrupt pin %c of device %s.%s\n",
626                  'A' + pin - 1, dev->slot_name, msg);
627       }
628  }
```

这里的 617 行表示：如果从寄存器 PCI_INTERRUPT_PIN 读出的数值非 0，就说明设备具有中断功能，因此调用 pcibios_lookup_irq()寻找其中断请求线的去向。要是这个函数正常返回，但 dev->irq 仍为 0，那就说明有问题了。

前面，在 pcibios_fixup_irqs()中也曾对所有设备逐个地调用过 pcibios_lookup_irq()，当时的第二个调用参数为 0，表示如果其中断请求线尚未连接就留着再说。而现在，则第二个调用参数为 1，表示若尚未连接就要选择一个合适的对象并完成连接。为什么要把这一步留到现在呢？一方面这是“ lazy computaion”，插在总线上的设备未必就是实际要用的，如果花费了代价而实际上并不使用是一种浪费，所以留到具体设备（也许是可安装模块）初始化的时候再来处理是合理的。另一方面，更为重要的是，当时在循环尚未结束之前并不知道系统中到底有多少设备连在同一条 PCI 中断请求线上，也不清楚中断控制器的各条输入线的负荷，因而不易作出真正合理的选择；而现在，则显然可以作出更合理的选择了。我们再看一下 pcibios_lookup_irq()有关的两个片段（arch/i386/kernel/pci-irq.c）：

==================== arch/i386/kernel/pci-irq.c 442 485 ====================

[pcibios_enable_device()>pcibios_enable_resources()>pcibios_lookup_irq()]

```
     /*
      * Find the best IRQ to assign: use the one
      * reported by the device if possible.
      */
     newirq = dev->irq;
     if (!newirq && assign) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
448         for (i = 0; i < 16; i++) {
449             if (!(mask & (1 << i)))
450                 continue;
451             if (pirq_penalty[i] < pirq_penalty[newirq] &&
452                !request_irq(i, pcibios_test_irq_handler, SA_SHIRQ, "pci-test", dev)) {
453                 free_irq(i, dev);
454                 newirq = i;
455             }
456         }
457     }
458     DBG(" -> newirq=%d", newirq);
459

477     } else if (newirq && r->set && (dev->class >> 8) != PCI_CLASS_DISPLAY_VGA) {
478         DBG(" -> assigning IRQ %d", newirq);
479         if (r->set(pirq_router_dev, dev, pirq, newirq)) {
480             eisa_set_level_irq(newirq);
481             DBG(" ... OK\n");
482             msg = "Assigned";
483             irq = newirq;
484         }
485     }
```

这两个片断的逻辑其实都是很简单的，读者在前面读过这个函数的其余部分，所以从总体上不应该有困难。但是，对 452 行却需要作一点说明。对于中断控制器一侧候选的中断请求输入线，这里通过 request_irq()（见第 3 章）试着登记一个空函数 pcibios_test_irq_handler()，如果成功再通过 free_irq() 将其撤销，这样就保证了以后登记真正的中断服务程序时也能成功。否则，如果试登记失败，就不能把这条输入线作为候选。代码中还调用了一个函数 eisa_set_level_irq()，那完全是因为硬件的特殊要求，我们就不关心了。

有些设备的情况还要更复杂，所以内核中又提供了更为一般化的 inline 函数 pci_module_init()（以及与之有关的一系列函数）来帮助设备驱动程序的设计和实现，读者可参阅本章第 9 节“通用串行外部总线 USB”。

8.5 块设备的驱动

块设备是文件系统的物质基础。而文件系统，则就是对块设备上所存储的内容按某种格式加以组织的结果。因此，对块设备（内容）的访问就有两种方式。一种是忽略对其内容的组织，即忽略文件系统的存在，而将其看成一个记录块的阵列（数组）；另一种则遵循文件系统的组织，将其看成通过各层目录组织在一起的文件集合，而每个文件则又是若干记录块的有序集合。由于“记录块数组”和“记录块的有序集合”都是有序的，在逻辑上又可以把它们看成线性的“字节流”。以书本为例，我们可以把一本书看成一叠页面，按页号找到其中的内容；也可以把它看成由若干章节构成，而按第几章第几

PDF-XCHANGE Click to buy NOW! www.docu-track.com

节找到其内容。同时，我们可以说第几页上的第几个字是什么，乃至整本书（假定每页上的字数是固定的）的第几个字是什么；也可以说第几章第几节的第几个字是什么。当引开一个代表块设备的文件节点，对这个“块设备文件”读 / 写时，我们把这设备（的内容）看作一个字节流，而忽略对其内容的组织即文件系统的存在。同样，当打开一个存在于块设备上的普通文件时，我们把这个特定的文件看作一个字节流，但是这个字节流按一定的格式和规则映射到块设备上。前者在 Unix 中称为“原始设备”，并且看成是字符设备，以示与后者的区别。这一点在 Linux 中有了改变，块设备还是块设备，只要是通过设备文件（节点）访问就是“原始”的。而在安装以后的块设备上，对普通文件的访问则自然不能忽略文件系统的组织了。这里还要说明，即使是对“原始”设备的访问，首先也必须在文件系统中找到代表着这个设备的文件节点，在这个过程中当然不能忽略文件系统的组织。

在“文件的读与写”一节中，读者已经看到对普通文件的访问怎样变换成了对块设备上记录块的访问，最后转化成对函数 ll_rw_block()的调用，从这个函数开始就进入了设备层。我们在这一节中继续往下看，不过在此之前还要先看一下对“原始”块设备文件的访问，包括打开文件以及对文件的读 / 写。从概念上说，这些内容或许应该放在“文件系统”一章中，但是因为设备文件的“文件层”很薄，其内容又与设备驱动关系很紧密，所以我们把它放在这里与设备驱动一起介绍。

先看块设备文件的打开。在“系统调用 mknod()”一节中读者已经看到，设备文件是通过 mknod()创建的，创建时将设备的类型（块设备或字符设备）、主设备号、次设备号都写入了代表该文件的索引节点中。这个索引节点本身也存在于一个块设备上，具体取决于该设备文件节点所在的文件系统。同时它又代表着一个块设备。二者可以相同，也可以不同。打开文件时，在 path_walk()中找到该设备文件节点所在的目录，再根据目录项的指引从文件系统所在的设备上读入这个索引节点，并检验它的模式，看看它所代表的是普通文件，还是目录、符号连接，或者是包括设备文件在内的特殊文件。就 Ext2 文件系统而言，这部分操作是和函数 ext2_read_inode()中进行的。我们已经在“从路径名到目标节点”一节中列出了这个函数的代码，读者可以回过去重读一下。对于包括设备文件在内的特殊文件，这个函数调用 init_special_inode()来初始化为这个文件创建的 inode 数据结构，这是在 fs/ext2/inode.c 中的第 1077 行执行的：

```
==================== fs/ext2/inode.c 961 962 ====================
961  void ext2_read_inode (struct inode * inode)
962  {
     . . . . . .
==================== fs/ext2/inode.c 1077 1078 ====================
1077          init_special_inode(inode, inode->i_mode,
1078                        le32_to_cpu(raw_inode->i_block[0]));
     . . . . . .
==================== fs/ext2/inode.c 1102 1102 ====================
1102 }
```

这里 inode 指向为该文件创建的 inode 结构，它的 i_mode 字段已经根据从设备上读入的索引节点中的有关信息设置好了，而 raw_inode 则指向从设备上读入的索引节点，即 ext2_inode 数据结构。我们已经在“系统调用 mknod()”一节中读过 init_special_inode()的代码，此处再列出其中有关块设备的几行（fs/devices.c）：

```
==================== fs/devices.c 200 201 ====================
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
void init_special_inode(struct inode *inode, umode_t mode, int rdev)
{
  . . . . . .
```

==================== fs/devices.c 206 210 ====================

```
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = to_kdev_t(rdev);
        inode->i_bdev = bdget(rdev);
    } else if (S_ISFIFO(mode))
  . . . . . .
```

==================== fs/devices.c 216 216 ====================

```
}
```

经过了这几行，inode 结构中的指针 i_fop 就指向了具体块设备的 file_operations 数据结构。同时，inode 结构中还有一个专用于块设备的指针 i_bdev，指向代表着具体块设备的 block_device 数据结构。这个数据结构或许已经存在于内存中，或许需要创建，由设备号惟一地加以确定。在"系统调用 mknod() "一节中已经列出了函数 bdget()的代码，块设备的 file_operations 数据结构定义见 fs/block_dev.c：

==================== fs/block_dev.c 709 717 ====================

```
struct file_operations def_blk_fops = {
    open:       blkdev_open,
    release:  blkdev_close,
    llseek:     block_llseek,
    read:       block_read,
    write:      block_write,
    fsync:      block_fsync,
    ioctl:      blkdev_ioctl,
};
```

这样，就为对 blkdev_open()的调用铺平了道路。当 path_walk()结束的时候，目标文件的 inode 结构已经与块设备的文件层挂上了钩。不过此时具体 block_device 结构中的指针 bd_op 还没有设置（如果是新创建的 block_device 结构），也就是还没有与具体块设备的操作挂上钩，那要到进入 blkdev_open()以后再来处理。所以这也是个走一步看一步，"摸着石头过河"的过程。

最后，在 dentry_open()中（见"文件的打开与关闭"）把目标文件 inode 结构中的指针 f_ops 复制到 file 结构中（f_op 指针），并通过相应 file_operations 结构中的函数指针 open 调用 blkdev_open()。这个函数的代码也在 fs/block_dev.c 中：

==================== fs/block_dev.c 644 664 ====================

[sys_open()>filp_open()>dentry_open()>blkdev_open()]

```
int blkdev_open(struct inode * inode, struct file * filp)
{
    int ret = -ENXIO;
    struct block_device *bdev = inode->i_bdev;
    down(&bdev->bd_sem);
    lock_kernel();
    if (!bdev->bd_op)
        bdev->bd_op = get_blkfops(MAJOR(inode->i_rdev));
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (bdev->bd_op) {
        ret = 0;
        if (bdev->bd_op->open)
            ret = bdev->bd_op->open(inode,filp);
        if (!ret)
            atomic_inc(&bdev->bd_openers);
        else if (!atomic_read(&bdev->bd_openers))
            bdev->bd_op = NULL;
    }
    unlock_kernel();
    up(&bdev->bd_sem);
    return ret;
}
```

每种具体的块设备都各有一套具体的操作，因而各自有一个类似于 file_operations 那样的数据结构，称为 block_device_operations，其定义见 include/linux/fs.h：

==================== include/linux/fs.h 760 766 ====================

```
struct block_device_operations {
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
    int (*ioctl) (struct inode *, struct file *, unsigned, unsigned long);
    int (*check_media_change) (kdev_t);
    int (*revalidate) (kdev_t);
};
```

如果说 file_operations 数据结构是连接虚拟的、抽象的 vfs 文件操作与具体文件系统（或文件类型）的文件操作之间的枢纽，那么 block_device_operations 就是连接抽象的块设备操作与具体块设备类型的操作之间的枢纽。块设备的类型是由主设备号惟一确定的，所以主设备号惟一地确定了一个具体的 block_device_operations 数据结构，而 blkdev_open()的任务就是根据主设备号找到相应的数据结构，并使 block_device 结构中的指针指向这个数据结构，然后调用由这个数据结构中的函数指针 open 所指向的函数（如果该指针不是 NULL）。如果某种设备在打开它的时候不需要做任何事，那么它的 block_device_operations 结构中的指针 open 就是 NULL。

寻找具体块设备类型的 block_device_operations 数据结构是由 get_blkfops()完成的，其代码在 fs/block_dev.c 中：

==================== fs/block_dev.c 487 507 ====================

[sys_open()>filp_open()>dentry_open()>blkdev_open()>get_blkfops()]

```
/*
    Return the function table of a device.
    Load the driver if needed.
*/
const struct block_device_operations * get_blkfops(unsigned int major)
{
    const struct block_device_operations *ret = NULL;

    /* major 0 is used for non-device mounts */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
496     if (major && major < MAX_BLKDEV) {
497 #ifdef CONFIG_KMOD
498         if (!blkdevs[major].bdops) {
499             char name[20];
500             sprintf(name, "block-major-%d", major);
501             request_module(name);
502         }
503 #endif
504         ret = blkdevs[major].bdops;
505     }
506     return ret;
507 }
```

内核中有一个数组 blkdevs[]，以主设备号为下标就可以通过数组中的表项找到各种设备的 block_device_operations 数据结构。这个数组也是在 fs/block_dev.c 中定义的：

==================== fs/block_dev.c 468 471 ====================

```
static struct {
    const char *name;
    struct block_device_operations *bdops;
} blkdevs[MAX_BLKDEV];
```

内核在初始化时根据系统的配置将有关块设备的block_device_operations结构连同设备名一起登记入数组中的相应表项。这样，get_blkfops()只要以主设备号为下标就可以找到给定设备的这个数据结构了（见代码中的第 504 行）。值得注意的是，如果内核支持可安装模块，那么即使内核在初始化时没有登记某种块设备，也可以在需要时通过 request_module()把用来支持该种设备的代码和数据结构安装进来。对这种模块的命名有个规则，那就是“ block-major-”加上具体设备的主设备号（见代码 500 行）。

在这里我们假定所用的块设备为 IDE 硬盘，因为这是最常用的。在 PC 机上最多可以有主、次（primary/secondary）两个 IDE 接口，每个 IDE 接口又可以支持主、从（master/slave）共两个 IDE 硬盘，所以最多可以有 4 个 IDE 硬盘（包括光盘），其中第一个 IDE 接口上主硬盘的主设备号为 3（其余硬盘的主设备号依次为 22、33 和 34）。IDE 硬盘都带有内装的控制器（IDE 为 Integrated Drives Electronics 的缩写），所以只需要“接口”而并不需要“控制卡”。在过去的十年中，这种硬盘有了很大的发展，以至于早期 IDE 硬盘所用的 block_device_operations 数据结构 hd_ops 已经不适用于新型的 IDE 硬盘，而只好另外定义新的数据结构 ide_fops，其定义在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 3492 3498 ====================

```
struct block_device_operations ide_fops[] = {{
    open:               ide_open,
    release:        ide_release,
    ioctl:              ide_ioctl,
    check_media_change:     ide_check_media_change,
    revalidate:         ide_revalidate_disk
}};
```

注意，这里说的是定义新的数据结构，而不是数据结构类型。之所以定义新的数据结构是因为有了一组新的函数。实际上，这里定义的是一个结构数组，只是数组中只有一个表项。这样就为将来预

留下进一步的发展空间。

回到 blkdev_open()的代码中。至此，我们已经把对抽象的“块设备文件”的操作逐层地具体化为对块设备的操作。但是“块设备”还是个抽象的概念，还需要进一步具体化，这就需要进入为 IDE 硬盘提供的操作了。现在，block-device 结构中的指针 bd_op 已经指向新型 IDE 硬盘的 block_device_operations 结构 ide_fops，而其中的指针 open 又指向 ide_open()，所以函数 blkdev_open()代码中的 655 行就通过这个指针调用 ide_open()，其代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 1832 1866 ====================

[sys_open()>filp_open()>dentry_open()>blkdev_open()>ide_open()]

```
static int ide_open (struct inode * inode, struct file * filp)
{
     ide_drive_t *drive;
     int rc;

     if ((drive = get_info_ptr(inode->i_rdev)) == NULL)
          return -ENXIO;
     MOD_INC_USE_COUNT;
     if (drive->driver == NULL)
          ide_driver_module();
#ifdef CONFIG_KMOD
     if (drive->driver == NULL) {
          if (drive->media == ide_disk)
               (void) request_module("ide-disk");
          if (drive->media == ide_cdrom)
               (void) request_module("ide-cd");
          if (drive->media == ide_tape)
               (void) request_module("ide-tape");
          if (drive->media == ide_floppy)
               (void) request_module("ide-floppy");
     }
#endif /* CONFIG_KMOD */
     while (drive->busy)
          sleep_on(&drive->wqueue);
     drive->usage++;
     if (drive->driver != NULL) {
          if ((rc = DRIVER(drive)->open(inode, filp, drive)))
               MOD_DEC_USE_COUNT;
          return rc;
     }
     printk ("%s: driver not present\n", drive->name);
     drive->usage--;
     MOD_DEC_USE_COUNT;
     return -ENXIO;
}
```

数据结构 ide_ops[]中提供了对 IDE 硬盘操作的函数跳转表，但是对具体 IDE 硬盘的操作还需要有关具体设备的许多数据，需要有个具体设备的“控制块”，这就是 ide_drive_t 数据结构，有关的定义在

include/linux/ide.h 中：

==================== include/linux/ide.h 257 352 ====================

```
/*
 * Now for the data we need to maintain per-drive:  ide_drive_t
 */

#define ide_scsi   0x21
#define ide_disk   0x20
#define ide_optical    0x7
#define ide_cdrom  0x5
#define ide_tape   0x1
#define ide_floppy 0x0

typedef union {
     unsigned all            : 8; /* all of the bits together */
     struct {
          unsigned set_geometry  : 1; /* respecify drive geometry */
          unsigned recalibrate   : 1; /* seek to cyl 0     */
          unsigned set_multmode  : 1; /* set multmode count */
          unsigned set_tune  : 1; /* tune interface for drive */
          unsigned reserved  : 4; /* unused */
     } b;
} special_t;

typedef struct ide_drive_s {
     request_queue_t          queue;   /* request queue */
     struct ide_drive_s      *next;    /* circular list of hwgroup drives */
     unsigned long sleep;         /* sleep until this time */
     unsigned long service_start;     /* time we started last request */
     unsigned long service_time; /* service time of last request */
     unsigned long timeout;       /* max time to wait for irq */
     special_t special; /* special action flags */
     byte     keep_settings;      /* restore settings after drive reset */
     byte     using_dma;      /* disk is using dma for read/write */
     byte     waiting_for_dma;    /* dma currently in progress */
     byte     unmask;         /* flag: okay to unmask other irqs */
     byte     slow;           /* flag: slow data port */
     byte     bswap;              /* flag: byte swap data */
     byte     dsc_overlap;        /* flag: DSC overlap */
     byte     nice1;              /* flag: give potential excess bandwidth */
     unsigned present   : 1; /* drive is physically present */
     unsigned noprobe   : 1; /* from:  hdx=noprobe */
     unsigned busy      : 1; /* currently doing revalidate_disk() */
     unsigned removable : 1; /* 1 if need to do check_media_change */
     unsigned forced_geom    : 1; /* 1 if hdx=c,h,s was given at boot */
     unsigned no_unmask : 1; /* disallow setting unmask bit */
     unsigned no_io_32bit    : 1; /* disallow enabling 32bit I/O */
     unsigned nobios         : 1; /* flag: do not probe bios for drive */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	unsigned revalidate	: 1; /* request revalidation */
	unsigned atapi_overlap	: 1; /* flag: ATAPI overlap (not supported) */
	unsigned nice0		: 1; /* flag: give obvious excess bandwidth */
	unsigned nice2		: 1; /* flag: give a share in our own bandwidth */
	unsigned doorlocking	: 1; /* flag: for removable only: door lock/unlock works */
	unsigned autotune	: 2; /* 1=autotune, 2=noautotune, 0=default */
	unsigned remap_0_to_1	: 2; /* 0=remap if ezdrive, 1=remap, 2=noremap */
	unsigned ata_flash	: 1; /* 1=present, 0=default */
	byte	scsi;		/* 0=default, 1=skip current ide-subdriver for ide-scsi emulation */
	byte	media;		/* disk, cdrom, tape, floppy, ... */
	select_t select;	/* basic drive/head select reg value */
	byte	ctl;	/* "normal" value for IDE_CONTROL_REG */
	byte	ready_stat;	/* min status value for drive ready */
	byte	mult_count;	/* current multiple sector setting */
	byte		mult_req;	/* requested multiple sector setting */
	byte		tune_req;	/* requested drive tuning setting */
	byte	io_32bit;	/* 0=16-bit, 1=32-bit, 2/3=32bit+sync */
	byte	bad_wstat;	/* used for ignoring WRERR_STAT */
	byte	nowerr;		/* used for ignoring WRERR_STAT */
	byte	sect0;		/* offset of first sector for DM6:DDO */
	byte		usage;		/* current "open()" count for drive */
	byte		head;		/* "real" number of heads */
	byte	sect;		/* "real" sectors per track */
	byte	bios_head;	/* BIOS/fdisk/LILO number of heads */
	byte	bios_sect;	/* BIOS/fdisk/LILO sectors per track */
	unsigned int	bios_cyl;	/* BIOS/fdisk/LILO number of cyls */
	unsigned int	cyl;	/* "real" number of cyls */
	unsigned long capacity;	/* total number of sectors */
	unsigned int	drive_data;	/* for use by tuneproc/selectproc as needed */
	void	*hwif;	/* actually (ide_hwif_t *) */
	wait_queue_head_t wqueue;	/* used to wait for drive in open() */
	struct hd_driveid *id;		/* drive model identification info */
	struct hd_struct  *part;	/* drive partition table */
	char	name[4];	/* drive name, such as "hda" */
	void		*driver;	/* (ide_driver_t *) */
	void	*driver_data;	/* extra driver data */
	devfs_handle_t	de;		/* directory for device */
	struct proc_dir_entry *proc;	/* /proc/ide/ directory entry */
	void	*settings;	/* /proc/ide/ drive settings */
	char	driver_req[10];	/* requests specific driver */
	int	last_lun;	/* last logical unit */
	int	forced_lun;	/* if hdxlun was given at boot */
	int	lun;	/* logical unit */
	int	crc_count;	/* crc counter to reduce drive speed */
	byte	quirk_list;	/* drive is considered quirky if set for a specific host */
	byte	suspend_reset;	/* drive suspend mode flag, soft-reset recovers */
	byte	init_speed;	/* transfer rate set at boot */
	byte	current_speed;	/* current transfer rate set */
	byte	dn;	/* now wide spread use */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
} ide_drive_t;
```

代码作者对结构中各个字段的意义和作用已经加了注释，以后随着代码的进展还会变得更加清楚。每个具体的 IDE 磁盘都有这么一个数据结构，函数 get_info_ptr()根据设备号找到这个数据结构并返回指向它的指针，这个函数的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 1628 1657 ====================
[sys_open()>filp_open()>dentry_open()>blkdev_open()>ide_open()>get_info_ptr()]

```
/*
 * get_info_ptr() returns the (ide_drive_t *) for a given device number.
 * It returns NULL if the given device number does not match any present drives.
 */
ide_drive_t *get_info_ptr (kdev_t i_rdev)
{
	int		major = MAJOR(i_rdev);
#if 0
	int		minor = MINOR(i_rdev) & PARTN_MASK;
#endif
	unsigned int  h;

	for (h = 0; h < MAX_HWIFS; ++h) {
		ide_hwif_t  *hwif = &ide_hwifs[h];
		if (hwif->present && major == hwif->major) {
			unsigned unit = DEVICE_NR(i_rdev);
			if (unit < MAX_DRIVES) {
				ide_drive_t *drive = &hwif->drives[unit];
#if 0
				if ((drive->present) && (drive->part[minor].nr_sects))
#else
				if (drive->present)
#endif
					return drive;
			}
			break;
		}
	}
	return NULL;
}
```

内核中有个数组 ide_hwifs[]，数组的每个元素都是一个 ide_hwif_t 数据结构，代表着系统中的一个可能的 IDE 接口（本节后面会给出这个数据结构的定义）。系统初始化时如果检测到一个 IDE 接口，就把相应表项中的 present 字段设置成 1。同时，这种数据结构中又有个 ide_drive_t 结构数组 drives[]。初始化时如果检测到某个接口上有磁盘相连，就将相应 ide_drive_t 结构中的 present 字段也设成 1，并根据检测到或从系统的 CMOS 芯片中读到的各项参数设置这个数据结构。也就是说，ide_hwif_t 数据结构是对 IDE 接口的描述，而 ide_drive_t 数据结构是对连接在具体 IDE 接口上的“ IDE 设备”的描述。例如，如果在系统的主（primary）IDE 接口上检测到有主 / 从两个磁盘相连，就把这两个磁盘的参数分别填入 ide_hwifs[0]中的 drives[0]和 drives[1]，并把它们的 present 字段设置成 1。再例如，如果在次

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

（secondary）IDE 接口上连接着一个 Mitsumi CDROM，那就把它的参数填入 ide_hwifs[1]里面的 drives[0]，并且把 ide_hwifs[1]中的字段 major 设置成 MITSUMI_CDROM_MAJOR。然后，当需要时，就由 get_info_ptr()根据主设备号在 ide_hwifs[]中搜索，找到相应的接口后再根据次设备号找到连接在该接口上的具体磁盘的 ide_drive_t 数据结构。

如果 get_info_ptr()找不到所要求的 ide_drive_t 结构，就说明系统中不存在相应的硬盘，所以 ide_open()返回出错代码-ENXIO，整个 open()操作也就失败了。

在 ide_drive_t 结构中（337 行）有个 void 指针 driver，可以根据不同的要求指向不同的 ide_driver_t 数据结构（注意 ide_driver_t 和 ide_drive_t 是两种不同的数据结构）。这个指针在系统初始化过程中检测到 IDE 接口上的设备时，根据设备的类型而设置成指向不同类型的数据结构。对于 IDE 硬盘，它指向一个 ide_driver_t 数据结构 idedisk_driver。同类的数据结构还有 idetape_driver、ide_cdrom_driver 以及 ide_floppy_driver，分别代表着可以连接到 IDE 接口上的不同类型的设备。

如果通过 get_info_ptr()找到了所需的 ide_drive_t 数据结构，但是它的 driver 指针却是 NULL，就说明初始化时虽然检测到了硬盘的存在，但是却因某种原因而未能完成对设备以及数据结构的初始化，所以在 ide_open()中调用 ide_driver_module()再试一次。

另一方面，如果内核支持可安装模块，那也可能是因为支持具体 IDE 设备的模块尚未安装或已被拆除而引起，所以试图根据具体的设备类型装入有关的模块。由 ide_driver_module()启动的操作是异步的，当前进程睡眠等待对设备的操作完成（1854～1855 行）以后再检查 ide_drive_t 结构中的指针 driver。如果已经变成非 0，就表示该设备仍旧与系统相连，并且已经成功地完成了初始化，所以可以通过 ide_driver_t 结构中的函数指针 open 启动特定设备的打开文件操作了。否则，要是 ide_drive_t 结构中的指针 driver 仍是 NULL，那就无法继续而失败了。

如上所述，IDE 硬盘的 ide_driver_t 数据结构是 idedisk_driver，其定义见 drivers/ide/ide-disk.c：

==================== drivers/ide/ide-disk.c 711 733 ====================

```
/*
 *   IDE subdriver functions, registered with ide.c
 */
static ide_driver_t idedisk_driver = {
     "ide-disk",          /* name */
     IDEDISK_VERSION,     /* version */
     ide_disk,       /* media */
     0,              /* busy */
     1,              /* supports_dma */
     0,              /* supports_dsc_overlap */
     NULL,                /* cleanup */
     do_rw_disk,          /* do_request */
     NULL,                /* end_request */
     NULL,                /* ioctl */
     idedisk_open,        /* open */
     idedisk_release,     /* release */
     idedisk_media_change,     /* media_change */
     idedisk_revalidate,       /* revalidate */
     idedisk_pre_reset, /* pre_reset */
     idedisk_capacity,  /* capacity */
```

```
        idedisk_special,   /* special */
        idedisk_proc       /* proc */
};
```

所以通过其指针 open 调用的函数（1858 行）为 idedisk_open()，其代码也在 drivers/ide/ide-disk.c 中：

==================== drivers/ide/ide-disk.c 478 492 ====================

[sys_open()>filp_open()>dentry_open()>blkdev_open()>ide_open()>idedisk_open()]

```
static int idedisk_open (struct inode *inode, struct file *filp, ide_drive_t *drive)
{
        MOD_INC_USE_COUNT;
        if (drive->removable && drive->usage == 1) {
                check_disk_change(inode->i_rdev);
                /*
                 * Ignore the return code from door_lock,
                 * since the open() has already succeeded,
                 * and the door_lock is irrelevant at this point.
                 */
                if (drive->doorlocking && ide_wait_cmd(drive, WIN_DOORLOCK, 0, 0, 0, NULL))
                        drive->doorlocking = 0;
        }
        return 0;
}
```

大部分 IDE 硬盘的介质都不是可拆卸的，所以这个函数在多数情况下并没有做什么事情。但是这并不意味着从总体上、从结构上说这一步是多余的。例如，对于同属 IDE 设备的 CDROM，相应的函数为 ide_cdrom_open()，那就是实实在在有事情要做的。

从 ide_open()正常返回，并且逐层返回到 dentry_open()中，整个打开块设备文件的操作就基本完成了。回顾一下这整个过程，就是从抽象的 vfs 层文件出发，逐层加以具体化，找到相应的数据结构并把这些数据结构联系在一起，层层打通关节的过程。

(1) file_operations 结构使 vfs 具体化成了特定的文件系统或文件类型（块设备文件）。

(2) block_device 数据结构使代表着抽象意义上的文件的 inode 结构具体化成了“块设备”。

(3) block_device_operations 结构使“块设备文件”操作进一步具体化成了“ IDE 设备”操作。

(4) ide_drive_t 结构将笼统的“ IDE 设备”具体化成了特定种类的 IDE 设备。

(5) ide_driver_t 结构将某种 IDE 设备的操作具体化成对特定 IDE 硬盘的操作。

我们以前讲过，打开文件的实质就是使一个进程与文件所代表的对象建立起连接。上述逐层具体化的过程实际上就是逐站选择前进方向，直至打通到达目标的道路。这个过程一经完成，就为进一步的文件操作做好了准备。读者也许会问，对普通文件的访问最终也要落实到具体的块设备上，但是为什么在那里就不需要打开具体的块设备文件呢？事实上，当把一个块设备安装到文件系统中去时，也已经走过了类似的路程，在文件系统中的安装点与具体块设备之间建立起了连接，只不过这种连接并不是建立在某个特定的进程与设备之间，并没有为之创建起 file 结构即文件操作的上下文而已。读者如果回过去看一下上册第 5 章“文件系统的安装与拆卸”一节，就会发现有些内容已经在那里提到过，只不过那时候我们的注意力不在块设备的细节，因而没有那么深入。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

现在文件已经打开，我们以系统调用 read()为例来看看对块设备的文件操作。我们知道，对于已经打开的块设备文件，其 file 结构中的 file_operations 结构指针指向 def_blk_fops，而从这个结构中可以发现其函数指针 read 指向 block_read()，其代码在 fs/block_dev.c 中：

==================== fs/block_dev.c 166 311 ====================

[sys_read()>block_read()]

```
166 ssize_t block_read(struct file * filp, char * buf, size_t count, loff_t *ppos)
167 {
168      struct inode * inode = filp->f_dentry->d_inode;
169      size_t block;
170      loff_t offset;
171      ssize_t blocksize;
172      ssize_t blocksize_bits, i;
173      size_t blocks, rblocks, left;
174      int bhrequest, uptodate;
175      struct buffer_head ** bhb, ** bhe;
176      struct buffer_head * buflist[NBUF];
177      struct buffer_head * bhreq[NBUF];
178      unsigned int chars;
179      loff_t size;
180      kdev_t dev;
181      ssize_t read;
182
183      dev = inode->i_rdev;
184      blocksize = BLOCK_SIZE;
185      if (blksize_size[MAJOR(dev)] && blksize_size[MAJOR(dev)][MINOR(dev)])
186           blocksize = blksize_size[MAJOR(dev)][MINOR(dev)];
187      i = blocksize;
188      blocksize_bits = 0;
189      while (i != 1) {
190           blocksize_bits++;
191           i >>= 1;
192      }
193
194      offset = *ppos;
195      if (blk_size[MAJOR(dev)])
196           size = (loff_t) blk_size[MAJOR(dev)][MINOR(dev)] << BLOCK_SIZE_BITS;
197      else
198           size = (loff_t) INT_MAX << BLOCK_SIZE_BITS;
199
200      if (offset > size)
201           left = 0;
202      /* size - offset might not fit into left, so check explicitly. */
203      else if (size - offset > INT_MAX)
204           left = INT_MAX;
205      else
206           left = size - offset;
207      if (left > count)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        left = count;
    if (left <= 0)
        return 0;
    read = 0;
    block = offset >> blocksize_bits;
    offset &= blocksize-1;
    size >>= blocksize_bits;
    rblocks = blocks = (left + offset + blocksize - 1) >> blocksize_bits;
    bhb = bhe = buflist;
    if (filp->f_reada) {
          if (blocks < read_ahead[MAJOR(dev)] / (blocksize >> 9))
            blocks = read_ahead[MAJOR(dev)] / (blocksize >> 9);
        if (rblocks > blocks)
            blocks = rblocks;

    }
    if (block + blocks > size) {
        blocks = size - block;
        if (blocks == 0)
            return 0;
    }

    /* We do this in a two stage process.  We first try to request
       as many blocks as we can, then we wait for the first one to
       complete, and then we try to wrap up as many as are actually
       done.  This routine is rather generic, in that it can be used
       in a filesystem by substituting the appropriate function in
       for getblk.

       This routine is optimized to make maximum use of the various
       buffers and caches. */

    do {
        bhrequest = 0;
        uptodate = 1;
        while (blocks) {
            --blocks;
            *bhb = getblk(dev, block++, blocksize);
            if (*bhb && !buffer_uptodate(*bhb)) {
                uptodate = 0;
                bhreq[bhrequest++] = *bhb;
            }

            if (++bhb == &buflist[NBUF])
                bhb = buflist;

            /* If the block we have on hand is uptodate, go ahead
               and complete processing. */
            if (uptodate)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                break;
            if (bhb == bhe)
                break;
        }

        /* Now request them all */
        if (bhrequest) {
            ll_rw_block(READ, bhrequest, bhreq);
        }

        do { /* Finish off all I/O that has actually completed */
            if (*bhe) {
                wait_on_buffer(*bhe);
                if (!buffer_uptodate(*bhe)) {   /* read error? */
                      brelse(*bhe);
                    if (++bhe == &buflist[NBUF])
                      bhe = buflist;
                    left = 0;
                    break;
                }
            }
            if (left < blocksize - offset)
                chars = left;
            else
                chars = blocksize - offset;
            *ppos += chars;
            left -= chars;
            read += chars;
            if (*bhe) {
                copy_to_user(buf,offset+(*bhe)->b_data,chars);
                brelse(*bhe);
                buf += chars;
            } else {
                while (chars-- > 0)
                    put_user(0,buf++);
            }
            offset = 0;
            if (++bhe == &buflist[NBUF])
                bhe = buflist;
        } while (left > 0 && bhe != bhb && (!*bhe || !buffer_locked(*bhe)));
        if (bhe == bhb && !blocks)
            break;
    } while (left > 0);

/* Release the read-ahead blocks */
    while (bhe != bhb) {
        brelse(*bhe);
        if (++bhe == &buflist[NBUF])
            bhe = buflist;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
306        };
307        if (!read)
308             return -EIO;
309        filp->f_reada = 1;
310        return read;
311   }
```

读者对这段代码也许会感到似曾相识，实际上它的主体确实与读普通文件时调用的函数 block_read_full_page()十分相似（见上册第 5 章“文件的写与读”一节）。开头部分关于记录块大小、关于预读的计算和处理等，读者在以前都看到过类似的代码。内核中有一些以块设备的主设备号为下标的指针数组，都是在 drivers/block/ll_rw_blk.c 中定义的。这些数组中的每个指针又指向另一个以次设备号为下标的数组，这样在逻辑上就等同于二维数组。但是每个以次设备号为下标的数组的大小却可以不同，所以比较节省空间。这里（186 行）用到的数组 blksize_size[][]中的内容为各个具体设备的记录块大小（见“文件系统的安装与拆卸”），而 blk_size[][]中的内容（196 行）则为各项具体设备中含有 1024 字节记录块的个数。所以，size 就是具体设备的总容量（以字节计）；而 left 则为该设备中剩下未读的字节数（206 行），208 行又进一步将其调整成本次读操作剩下未读的字节数。212 行根据位移计算出起始块号，然后 213 行把整个设备内的位移调整为记录块内的位移，215 行又计算出本次读操作中要读的块数。最后，再基于预读的考虑和设备的总容量，再对本次操作中要读出的记录块数 blocks 作出调整。

接着就是针对本次读文件操作中要读出的每个记录块的 do-while 循环了。函数 getblk()根据设备号与记录块号的杂凑值试图从相应的杂凑表队列中找到该记录块的缓冲区。如果找不到，就为之分配一个缓冲区并将其指针填入一个 buffer_head 结构指针数组 bhreq[]，准备从设备上读入这个记录块。我们已经在“文件的写与读”一节中读过这个函数的代码，这里不再重复，读者不妨回过去重温一下。这个函数根据目标设备的设备号和设备上的逻辑记录块号，先在记录块的杂凑表队列中寻找，如果找到就简单了，不用从设备上读入了。否则就要为之分配一个空闲的 buffer_head 数据结构连同大小相符的缓冲区（实际上是缓冲页面的一部分）。

在“文件的写和读”一节中我们讲过，文件的内容在内存中既按记录块缓冲，又按页面缓冲。文件的 inode 结构中维持着一个缓冲页面队列，当需要读入新的页面时就通过 create_page_buffers()分配一个存储页面，再把这个页面划分成若干缓冲区，各自相当于一个记录块，同时分配相应数量的 buffer_head 数据结构，并使它们指向这些缓冲区。每个缓冲区都有个 buffer_head 数据结构，定义于 include/linux/fs.h：

==================== include/linux/fs.h 209 249 ====================

```
/*
 * Try to keep the most commonly used fields in single cache lines (16
 * bytes) to improve performance.  This ordering should be
 * particularly beneficial on 32-bit processors.
 *
 * We use the first 16 bytes for the data which is used in searches
 * over the block hash lists (ie. getblk() and friends).
 *
 * The second 16 bytes we use for lru buffer scans, as used by
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * sync_buffers() and refill_freelist().  -- sct
 */
struct buffer_head {
	/* First cache line: */
	struct buffer_head *b_next; /* Hash queue list */
	unsigned long b_blocknr;    /* block number */
	unsigned short b_size;      /* block size */
	unsigned short b_list;      /* List that this buffer appears */
	kdev_t b_dev;           /* device (B_FREE = free) */

	atomic_t b_count;       /* users using this block */
	kdev_t b_rdev;              /* Real device */
	unsigned long b_state;      /* buffer state bitmap (see above) */
	unsigned long b_flushtime;  /* Time when (dirty) buffer should be written */

	struct buffer_head *b_next_free;/* lru/free list linkage */
	struct buffer_head *b_prev_free;/* doubly linked list of buffers */
	struct buffer_head *b_this_page;/* circular list of buffers in one page */
	struct buffer_head *b_reqnext;   /* request queue */

	struct buffer_head **b_pprev;    /* doubly linked list of hash-queue */
	char * b_data;              /* pointer to data block (512 byte) */
	struct page *b_page;        /* the page this bh is mapped to */
	void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
	void *b_private;        /* reserved for b_end_io */

	unsigned long b_rsector;    /* Real buffer location on disk */
	wait_queue_head_t b_wait;

	struct inode *          b_inode;
	struct list_head     b_inode_buffers; /* doubly linked list of inode dirty buffers */
};
```

结构中的指针 b_data 指向真正的“缓冲区”，即相应内存页面中的一部分。每个缓冲区通过其 buffer_head 数据结构链入到若干队列中：

(1) 通过指针 b_next 和双重指针 b_pprev 链入一个单链杂凑队列；

(2) 通过 b_next_free 和 b_prev_free 链入一个双链的 LRU 队列或空闲队列，字段 b_list 则说明目前缓冲区在哪一个 LRU 队列中；

(3) 通过指针 b_this_page 将属于同一页面的缓冲区链在一起，并通过指针 b_page 指向所属页面的 page 数据结构；

(4) 通过队列头 b_inode_buffers 链入所属文件的 inode 数据结构，并通过指针 b_inode 指向该 inode 数据结构。

此外，结构中的 b_dev 为目标设备的设备号；这个字段是在通过 getblk()分配缓冲区时根据调用参数设置的，从 block_read()的代码（183 和 245 行）可以看出这个设备号来自 inode 结构中的 i_rdev，那就是 inode 结构所代表文件所在的设备；当所代表的文件为块设备文件时，那就是目标设备的设备号。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

有时候目标设备只是个逻辑设备，最后要落实到另一个设备上，此时用另一个字段 b_rdev 用于实际设备的设备号。

对普通文件而言，在文件层上的操作着眼于页面。相比之下，对块设备文件的访问则直接就跟缓冲区打交道，而跳过了页面这一层。所以直接就通过 getblk()搜索或分配已经有了缓冲区的 buffer_head 数据结构。相比之下，getblk()是个比较低层的函数，主要用于块设备上为文件系统的组织和管理所需的记录块，即所谓“ meta data”，也可以说是供文件系统“内部使用”。例如，在对普通文件的读写中，如果文件比较大而需要间接映射，那就需要找到用于间接映射表的记录块，这就要调用 getblk()了（见“文件的写和读”一节中 ext2_alloc_branch()的代码）。又如，在 path_walk()中要读入一个目录，即与一个目录节点相联系的记录块时，也要调用 getblk()。这些记录块在逻辑上都不属于任何一个普通文件的内容，所以相应的存储页面不挂入任何 inode 结构中的缓冲页面队列。

回到 block_read()的代码中，看一下调用 getblk()时（245 行）使用的参数，就可以发现：作为设备上逻辑块号的 block 正是前面通过线性映射得到的文件内逻辑块号。也就是说二者是相同的。然而，对于普通文件，从文件内逻辑块号到设备上逻辑块号的映射却是相当复杂的。这正是块设备文件的文件层相比之下显得很单薄、很简单的根本原因。

从 getblk()返回到 block_read()以后，就可以知道是否真的需要从块设备上读入某个记录块了。但是，每次只从块设备上读入一个记录块是很不经济的。块设备通常是磁盘设备，它的介质在旋转，并且划分成“柱面”，在读出之前要先将磁头定位。由于块设备的这些特性，成片地读出逻辑上连续或邻近的记录块比分次读出个别的记录块有效得多。所以，block_read()通过数组 bhreq[]积累起成片读入请求，具体的办法是：

(1) 如果下一个记录块，即本次 do-while 循环中的第一个记录块，不需要从设备上读入，那么这个 while 循环马上就结束了（257 行），此时 bhrequest 为 0，所以跳过 ll_rw_block()不从设备上读入。数组 buflist[]用来收集和积累所需的记录块缓冲区（见 216 行和 245 行），其中有些不需要从设备上读入（如果 246 行的条件能满足），有些则需要从设备上读入。
(2) 反之，如果本次 do-white 循环中的第一个记录块需要从设备上读入，那就要通过 bhreq[]积累起成片的读入请求，但不超过数组 buflist[]的大小。第 243 行至 260 行的 while 循环的作用，就是在数组 buflist[]中积累起需要从块设备读入的记录块缓冲区，同时在 bhreq[]中积累起成片的读入请求，直至把 buflist[]填满或者 blocks 变成了 0 为止。对数组 buflist[]是按循环缓冲区的方式使用的（见 251～252 行以及 258～259 行），每次进入 243 行的 while 循环时 bhb 和 bhe 相同，开始循环以后，当二者再次相同时，就说明 buflist[]中已经填满了（258 行）。积累了成片的记录块缓冲区以后，就通过 ll_rw_block()启动块设备的成片读入。数组 bhreq[]的大小与 buflist[]相同，在最坏的情况下所有的缓冲区都需要从设备上读入。但也可能有几个缓冲区不需要读入，从而不出现在 bhreq[]中。
(3) 然后，通过一个内层 do-while 循环（267 行）处理 buflist[]中积累起的下一片缓冲区，处理完以后又回到外层 do-while 循环的开头（240 行），直至全部完成。

至此，对块设备文件和对普通文件的读操作终于殊途同归，都归结成了对 ll_rw_block()的调用，这才真正进入了设备驱动层。顾名思义，ll_rw_block()的作用是“低层块设备读写”。

类似地，对块设备的写操作 block_write()最终也是对 ll_rw_block()的调用。不过块设备的写操作常常需要先从设备上读入一个记录块，加以修改以后再把它写回去。我们把 block_write()的代码留给读者。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在“文件的写和读”一节中，读者已经看到对普通文件的写操作并不直接启动块设备的写操作，而是在改变了记录块缓冲区的内容后就将其提交给一个内核线程 kflushd（其执行程序为 bdflush()）。这个线程平时都在睡眠。一旦需要“冲刷”到块设备上的页面积累到一定的数量（以及在其它一些条件下）时就将它唤醒，使它执行一个函数 flush_dirty_buffers()。可想而知，flush_dirty_buffers()也调用 ll_rw_block()启动对具体块设备的写操作。函数 flush_dirty_buffers()的代码在文件 fs/buffer.c 中：

==================== fs/buffer.c 2537 2589 ====================

[bdflush()>flush_dirty_buffers()]

```
/* This is the _only_ function that deals with flushing async writes
   to disk.
   NOTENOTENOTENOTE: we _only_ need to browse the DIRTY lru list
   as all dirty buffers lives _only_ in the DIRTY lru list.
   As we never browse the LOCKED and CLEAN lru lists they are infact
   completly useless. */
static int flush_dirty_buffers(int check_flushtime)
{
    struct buffer_head * bh, *next;
    int flushed = 0, i;

 restart:
    spin_lock(&lru_list_lock);
    bh = lru_list[BUF_DIRTY];
    if (!bh)
         goto out_unlock;
    for (i = nr_buffers_type[BUF_DIRTY]; i-- > 0; bh = next) {
         next = bh->b_next_free;

         if (!buffer_dirty(bh)) {
              __refile_buffer(bh);
              continue;
         }
         if (buffer_locked(bh))
              continue;

         if (check_flushtime) {
              /* The dirty lru list is chronologically ordered so
                 if the current bh is not yet timed out,
                 then also all the following bhs
                 will be too young. */
              if (time_before(jiffies, bh->b_flushtime))
                   goto out_unlock;
         } else {
              if (++flushed > bdf_prm.b_un.ndirty)
                   goto out_unlock;
         }

         /* OK, now we are committed to write it out. */
         atomic_inc(&bh->b_count);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
2577            spin_unlock(&lru_list_lock);
2578            ll_rw_block(WRITE, 1, &bh);
2579            atomic_dec(&bh->b_count);
2580
2581            if (current->need_resched)
2582                schedule();
2583            goto restart;
2584        }
2585  out_unlock:
2586        spin_unlock(&lru_list_lock);
2587
2588        return flushed;
2589 }
```

系统中有个缓冲区队列的数组 lru_list[]，这个数组以缓冲区的类型为下标，所以 lru_list[BUF_DIRTY]就是其中已经“脏”了的缓冲区的队列。除此以外，数组中还有 BUF_CLEAN、BUF_LOCKED、BUF_PROTECTED 等队列。所有的缓冲区都按 LRU，即最后一次受到访问的时间先后排在其中的某个队列中。另一个数组 nr_buffers_type[]与之对应，记录着每个队列的大小。这个函数扫描脏缓冲区队列，依次考察队列中的页面。如果一个页面虽然在 BUF_DIRTY 队列中，但是实际上已经不再是“脏”的了，那就通过__refile_buffer()把它转移到其它队列中。如果一个缓冲区已经加了锁就把它跳过。否则，就调用 ll_rw_block()把它写入设备，写入设备以后这个缓冲区就变成干净的了。不过，这种操作不一定是对队列中所有缓冲区的，如果参数 check_flushtime 非 0，则表示仅对已经超时的缓冲区进行处理。由于队列中所有的缓冲区都是按“年龄”排列的，所以在碰到第一个尚未超时的缓冲区时循环就结束了。还有，由于这整个过程可能需要较长的时间，也不一定是在系统调用中执行，所以在每次调用 ll_rw_block()以后都要检查是否需要调度（通常都是从系统空间返回到用户空间的前夕才检查），如果需要就主动调用 schedule()加以调度。以后当再次被调度运行时则回到 restart 处（2548 行）重新扫描整个队列，因为情况可能已经改变。

此外，系统中还有一个内核线程 kupdate()，它管的事更宽，包括超级块的同步、索引节点的同步，也包括缓冲区的同步（即冲刷），还包括对 tq_disk 任务队列（见第 3 章）的执行。它也通过 flush_dirty_buffers()冲刷内容已经改变的缓冲区，所以最终也是调用 ll_rw_block()。

由此可见，无论是对普通文件的读 / 写，还是对块设备文件的读 / 写，最终都是通过 ll_rw_block()完成的。与这个函数并立的还有 ll_rw_block_locked()，这两个函数的代码在 drivers/block/ll_rw_blk.c 中。这里要再次强调，对 ll_rw_block()的调用路径是比较多的，我们随同代码列出的只是其中之一。

==================== drivers/block/ll_rw_blk.c 987 1093 ====================

[sys_read()>block_read()>ll_rw_block()]

```
987  /**
988   * ll_rw_block: low-level access to block devices
989   * @rw: whether to %READ or %WRITE or maybe %READA (readahead)
990   * @nr: number of &struct buffer_heads in the array
991   * @bhs: array of pointers to &struct buffer_head
992   *
993   * ll_rw_block() takes an array of pointers to &struct buffer_heads,
994   * and requests an I/O operation on them, either a %READ or a %WRITE.
```

```
 * The third %READA option is described in the documentation for
 * generic_make_request() which ll_rw_block() calls.
 *
 * This function provides extra functionality that is not in
 * generic_make_request() that is relevant to buffers in the buffer
 * cache or page cache.  In particular it drops any buffer that it
 * cannot get a lock on (with the BH_Lock state bit), any buffer that
 * appears to be clean when doing a write request, and any buffer that
 * appears to be up-to-date when doing read request.  Further it marks
 * as clean buffers that are processed for writing (the buffer cache
 * wont assume that they are actually clean until the buffer gets
 * unlocked).
 *
 * ll_rw_block sets b_end_io to simple completion handler that marks
 * the buffer up-to-date (if approriate), unlocks the buffer and wakes
 * any waiters.  As client that needs a more interesting completion
 * routine should call submit_bh() (or generic_make_request())
 * directly.
 *
 * Caveat:
 *  All of the buffers must be for the same device, and must also be
 *  of the current approved size for the device.  */

void ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
{
	unsigned int major;
	int correct_size;
	int i;

	major = MAJOR(bhs[0]->b_dev);

	/* Determine correct block size for this device. */
	correct_size = BLOCK_SIZE;
	if (blksize_size[major]) {
		i = blksize_size[major][MINOR(bhs[0]->b_dev)];
		if (i)
			correct_size = i;
	}

	/* Verify requested block sizes. */
	for (i = 0; i < nr; i++) {
		struct buffer_head *bh;
		bh = bhs[i];
		if (bh->b_size != correct_size) {
			printk(KERN_NOTICE "ll_rw_block: device %s: "
			       "only %d-char blocks implemented (%u)\n",
			       kdevname(bhs[0]->b_dev),
			       correct_size, bh->b_size);
			goto sorry;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
		}
	}

	if ((rw & WRITE) && is_read_only(bhs[0]->b_dev)) {
		printk(KERN_NOTICE "Can't write to read-only device %s\n",
		       kdevname(bhs[0]->b_dev));
		goto sorry;
	}

	for (i = 0; i < nr; i++) {
		struct buffer_head *bh;
		bh = bhs[i];

		/* Only one thread can actually submit the I/O. */
		if (test_and_set_bit(BH_Lock, &bh->b_state))
			continue;

		/* We have the buffer lock */
		bh->b_end_io = end_buffer_io_sync;

		switch(rw) {
		case WRITE:
			if (!atomic_set_buffer_clean(bh))
				/* Hmmph! Nothing to write */
				goto end_io;
			__mark_buffer_clean(bh);
			break;

		case READA:
		case READ:
			if (buffer_uptodate(bh))
				/* Hmmph! Already have it */
				goto end_io;
			break;
		default:
			BUG();
	end_io:
			bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
			continue;
		}

		submit_bh(rw, bh);
	}
	return;

sorry:
	/* Make sure we don't get infinite dirty retries.. */
	for (i = 0; i < nr; i++)
		mark_buffer_clean(bhs[i]);
```

```
}
```

参数 bhs[]为一个缓冲区头部 buffer_head 的指针数组，数组中的每个指针都指向一个需要从设备读／写的记录块的 buffer_head 数据结构；参数 nr 则为该数组的大小。数组中指定要读／写的记录块不必是连续的（不过通常是邻近的），但是必须在同一设备上。参数 rw 指明了要进行的操作。

每种块设备就好像是个服务器，都有一个操作请求队列，队列的头部是一个 request_queue_t 数据结构，有关的定义见 include/linux/blkdev.h：

==================== include/linux/blkdev.h 74 125 ====================

```
struct request_queue
{
	/*
	 * the queue request freelist, one for reads and one for writes
	 */
	struct list_head	request_freelist[2];

	/*
	 * Together with queue_head for cacheline sharing
	 */
	struct list_head	queue_head;
	elevator_t		elevator;

	request_fn_proc		* request_fn;
	merge_request_fn	* back_merge_fn;
	merge_request_fn	* front_merge_fn;
	merge_requests_fn	* merge_requests_fn;
	make_request_fn		* make_request_fn;
	plug_device_fn		* plug_device_fn;
	/*
	 * The queue owner gets to use this for whatever they like.
	 * ll_rw_blk doesn't touch it.
	 */
	void			* queuedata;

	/*
	 * This is used to remove the plug when tq_disk runs.
	 */
	struct tq_struct	plug_tq;

	/*
	 * Boolean that indicates whether this queue is plugged or not.
	 */
	char			plugged;

	/*
	 * Boolean that indicates whether current_request is active or
	 * not.
	 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
113      char          head_active;
114
115      /*
116       * Is meant to protect the queue in the future instead of
117       * io_request_lock
118       */
119      spinlock_t         request_lock;
120
121      /*
122       * Tasks wait here for free request
123       */
124      wait_queue_head_t  wait_for_request;
125  };
```

===================== include/linux/blkdev.h 11 11 =====================

```
11   typedef struct request_queue request_queue_t;
```

数据结构中各个字段的作用和意义随着代码的进展会变得清楚起来。结构中从 request_fn 到 plug_device_fn 都是一些函数指针，例如第 87 行的意思是说：request_fn 是一个指针，指向类型为 request_fn_proc 的对象。而 request_fn_proc 则通过#typedef 定义为一种函数（见 include/linux/blkdev.h）：

===================== include/linux/blkdev.h 63 63 =====================

```
63   typedef void (request_fn_proc) (request_queue_t *q);
```

其余的函数指针也与此类似，这些指针（连同其它字段）都是在相应块设备初始化时设置好的。需要对一个块设备进行操作时，就为之设置好一个数据结构（见后）并将其挂入相应的请求队列。

为了把各种块设备的操作请求队列有效地组织起来，内核中还设置了一个结构数组 blk_dev[]，见 drivers/block/ll_rw_blk.c：

===================== drivers/block/ll_rw_blk.c 72 76 =====================

```
72   /* blk_dev_struct is:
73    *    *request_fn
74    *    *current_request
75    */
76   struct blk_dev_struct blk_dev[MAX_BLKDEV]; /* initialized by blk_dev_init() */
```

这个数组以主设备号为下标，数组中的每个元素都是一个 blk_dev_struct 数据结构，其定义在 include/linux/blkdev.h 中给出：

===================== include/linux/blkdev.h 127 134 =====================

```
127  struct blk_dev_struct {
128      /*
129       * queue_proc has to be atomic
130       */
131      request_queue_t         request_queue;
132      queue_proc         *queue;
133      void          *data;
134  };
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

它的主体就是（操作）请求队列 request_queue。此外，结构中还有一个函数指针 queue，当这个指针为非 0 时就调用这个函数来找到具体设备的请求队列，这是考虑具有同一主设备号的多项设备而设的。这个指针也在设备初始化时设置好，通常当该指针非 0 时还要使用数据结构中的另一个指针 data 来提供辅助性的信息，以帮助该函数找到特定设备的请求队列。

进入 ll_rw_block()以后，先对记录块大小作一些检查；然后，如果是写访问，则还要检查目标设备是否可写。内核中有个二维数组 ro_bits，定义于 drivers/block/ll_rw_blk.c：

==================== drivers/block/ll_rw_blk.c 538 538 ====================

```
static long ro_bits[MAX_BLKDEV][8];
```

每个设备在这数组中都有个标志位，通过系统调用 ioctl()可以将一个标志位设置成 1 或 0，表示相应设备为只读或可写，而 is_read_only()则根据设备号检查这个数组中的标志位是否为 1。

接下去，就是通过 1053 行的 for 循环依次处理对各个缓冲区的读写请求了。对于要读写的每个记录块，首先将其缓冲区加上锁，还要将其 buffer_head 结构中的函数指针 b_end_io 设置成指向 end_buffer_io_sync()。当完成对给定记录块的读写时，就调用这个函数。此外，对于待写的缓冲区，其 BH_Dirty 标志位应该是 1，否则就不需要写了，而既然写了就要把它清成 0，并通过__mark_buffer_clean()将缓冲区转移到干净页面的 LRU 队列中。反之，对于待读的缓冲区，则其 Uptodate 标志位应该是 0，否则就不需要读了。每个具体的设备就好像是个服务器，所以最后具体的读写是通过 submit_bh()将读写请求提交给“服务器”完成的，每次一个记录块，其代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 939 976 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()]

```
/**
 * submit_bh: submit a buffer_head to the block device later for I/O
 * @rw: whether to %READ or %WRITE, or mayve to %READA (read ahead)
 * @bh: The &struct buffer_head which describes the I/O
 *
 * submit_bh() is very similar in purpose to generic_make_request(), and
 * uses that function to do most of the work.
 *
 * The extra functionality provided by submit_bh is to determine
 * b_rsector from b_blocknr and b_size, and to set b_rdev from b_dev.
 * This is is appropriate for IO requests that come from the buffer
 * cache and page cache which (currently) always use aligned blocks.
 */
void submit_bh(int rw, struct buffer_head * bh)
{
     if (!test_bit(BH_Lock, &bh->b_state))
          BUG();

     set_bit(BH_Req, &bh->b_state);

     /*
      * First step, 'identity mapping' - RAID or LVM might
      * further remap this.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
962       */
963      bh->b_rdev = bh->b_dev;
964      bh->b_rsector = bh->b_blocknr * (bh->b_size>>9);
965
966      generic_make_request(rw, bh);
967
968      switch (rw) {
969           case WRITE:
970                kstat.pgpgout++;
971                break;
972           default:
973                kstat.pgpgin++;
974                break;
975      }
976  }
```

对具体记录块缓冲区的操作请求必须是独占的，所以在调用 submit_bh()之前必须已经对缓冲区通过其 BH_LOCK 标志位加了锁（见前面的 1058 行），这里则通过检验该标志位来加以验证。进一步，由一个进程提交的操作请求未完成之前，是不允许其它进程再来提交对同一记录块缓冲区的操作请求的。所以，在提交对一特定记录块缓冲区的操作请求之前，还要通过标志位 BH_Req 标志位加锁。

显然，操作的主体是 generic_make_request()。对于一般的设备，实际的目标设备 bh->b_rdev 就是逻辑的目标设备 bh->b_dev，而设备上的扇区号 bh->b_rsector 则要根据目标块号 bh->b_blocknr 换算，因为扇区的大小通常为 512 字节，而记录块的大小为 1024 字节。值得注意的是，在文件系统层次上这里的记录块号已经是所谓“物理块号”了，但是到了设备驱动层这仍是逻辑意义上的操作对象，而物理意义上的对象是扇区。

函数 generic_make_request()的代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 850 936 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()]

```
850  /**
851   * generic_make_request: hand a buffer head to it's device driver for I/O
852   * @rw:  READ, WRITE, or READA - what sort of I/O is desired.
853   * @bh:  The buffer head describing the location in memory and on the device.
854   *
855   * generic_make_request() is used to make I/O requests of block
856   * devices. It is passed a &struct buffer_head and a &rw value.  The
857   * %READ and %WRITE options are (hopefully) obvious in meaning.  The
858   * %READA value means that a read is required, but that the driver is
859   * free to fail the request if, for example, it cannot get needed
860   * resources immediately.
861   *
862   * generic_make_request() does not return any status.  The
863   * success/failure status of the request, along with notification of
864   * completion, is delivered asynchronously through the bh->b_end_io
865   * function described (one day) else where.
866   *
867   * The caller of generic_make_request must make sure that b_page,
```

```
 * b_addr, b_size are set to describe the memory buffer, that b_rdev
 * and b_rsector are set to describe the device address, and the
 * b_end_io and optionally b_private are set to describe how
 * completion notification should be signaled.  BH_Mapped should also
 * be set (to confirm that b_dev and b_blocknr are valid).
 *
 * generic_make_request and the drivers it calls may use b_reqnext,
 * and may change b_rdev and b_rsector.  So the values of these fields
 * should NOT be depended on after the call to generic_make_request.
 * Because of this, the caller should record the device address
 * information in b_dev and b_blocknr.
 *
 * Apart from those fields mentioned above, no other fields, and in
 * particular, no other flags, are changed by generic_make_request or
 * any lower level drivers.
 * */
void generic_make_request (int rw, struct buffer_head * bh)
{
    int major = MAJOR(bh->b_rdev);
    request_queue_t *q;

    if (!bh->b_end_io) BUG();
    if (blk_size[major]) {
        unsigned long maxsector = (blk_size[major][MINOR(bh->b_rdev)] << 1) + 1;
        unsigned int sector, count;

        count = bh->b_size >> 9;
        sector = bh->b_rsector;

        if (maxsector < count || maxsector - count < sector) {
            bh->b_state &= (1 << BH_Lock) | (1 << BH_Mapped);
            if (blk_size[major][MINOR(bh->b_rdev)]) {

                /* This may well happen - the kernel calls bread()
                   without checking the size of the device, e.g.,
                   when mounting a device. */
                printk(KERN_INFO
                      "attempt to access beyond end of device\n");
                printk(KERN_INFO "%s: rw=%d, want=%d, limit=%d\n",
                      kdevname(bh->b_rdev), rw,
                      (sector + count)>>1,
                      blk_size[major][MINOR(bh->b_rdev)]);
            }
            bh->b_end_io(bh, 0);
            return;
        }
    }

    /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
917         * Resolve the mapping until finished. (drivers are
918         * still free to implement/resolve their own stacking
919         * by explicitly returning 0)
920         */
921        /* NOTE: we don't repeat the blk_size check for each new device.
922         * Stacking drivers are expected to know what they are doing.
923         */
924        do {
925             q = blk_get_queue(bh->b_rdev);
926             if (!q) {
927                  printk(KERN_ERR
928                        "generic_make_request: Trying to access nonexistent block-device %s (%ld)\n",
929                        kdevname(bh->b_rdev), bh->b_rsector);
930                  buffer_IO_error(bh);
931                  break;
932             }
933
934        }
935        while (q->make_request_fn(q, rw, bh));
936   }
```

如前所述，数组 blk_size[][]中的内容为各项具体设备中含有 1024 字节记录块的个数。这个数组是在 drivers/block/ll_rw_blk.c 中定义的：

==================== drivers/block/ll_rw_blk.c 78 86 ====================

```
78   /*
79    * blk_size contains the size of all block-devices in units of 1024 byte
80    * sectors:
81    *
82    * blk_size[MAJOR][MINOR]
83    *
84    * if (!blk_size[MAJOR]) then no minor size checking is done.
85    */
86   int * blk_size[MAX_BLKDEV];
```

注意 blk_size[]实际上是个 int 指针数组，以主设备号为下标，如果某个主设备号所对应的设备存在就指向一个以次设备号为下标的 int 数组，否则就为 0。用这样的方式实现“稀疏”数组，即有很多元素为 0 的数组，可以大大节省空间，而运行效率的下降却并不显著，所以在 Linux 内核中常常要用到。代码中的 890～914 行检查记录块的扇区范围，记录块的起点扇区号为 bh->b_rsector，扇区的个数等于记录块的大小 bh->b_size 除以 512。如果越出了范围就通过 printk()在系统的运行日志中登记出错信息，然后返回。

接着要根据设备号找到目标设备的操作请求队列，这是由 blk_get_queue()完成的，其代码在 drivers/block/ll_rw_blk.c 中。

==================== drivers/block/ll_rw_blk.c 138 152 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>blkget_queue()]

```
138  /*
```

```
 * NOTE: the device-specific queue() functions
 * have to be atomic!
 */
request_queue_t *blk_get_queue(kdev_t dev)
{
     request_queue_t *ret;
     unsigned long flags;

     spin_lock_irqsave(&io_request_lock,flags);
     ret = __blk_get_queue(dev);
     spin_unlock_irqrestore(&io_request_lock,flags);

     return ret;
}
```

==================== drivers/block/ll_rw_blk.c 128 136 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>blkget_queue()>__blk_get_queue()]

```
static inline request_queue_t *__blk_get_queue(kdev_t dev)
{
     struct blk_dev_struct *bdev = blk_dev + MAJOR(dev);

     if (bdev->queue)
          return bdev->queue(dev);
     else
          return &blk_dev[MAJOR(dev)].request_queue;
}
```

以主设备号为下标，就可以在 blk_dev[]中找到目标设备的 blk_dev_struct 结构。对一种设备第一次调用__blk_get_queue()时，要通过其函数指针 queue 找到这种设备的读写请求队列，找到了以后就把它记录在结构中的 request_queue 字段，以后就简单了。对于 IDE 硬盘，其 blk_dev_struct 结构中的函数指针 queue 在初始化时设置成指向 ide_get_queue()，而且其指针 data 设置成指向代表着相应硬件接口的 ide_hwif_t 数据结构。所以，对于 IDE 硬盘，将会调用 ide_get_queue()取得其操作请求队列，这个函数的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 1367 1375 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>blkget_queue()>__blk_get_queue()>ide_get_queue()]

```
/*
 * ide_get_queue() returns the queue which corresponds to a given device.
 */
request_queue_t *ide_get_queue (kdev_t dev)
{
     ide_hwif_t *hwif = (ide_hwif_t *)blk_dev[MAJOR(dev)].data;

     return &hwif->drives[DEVICE_NR(dev) & 1].queue;
}
```

这里的宏操作 DEVICE_NR(dev)从设备号 dev 中取出次设备号，然后再取其最低位，这保证了次设备号只能取 0 和 1 两种值。每种 IDE 设备，更确切地说是每个 IDE 接口，都有个 ide_hwif_t 数据结

PDF-XCHANGE Click to buy NOW! www.docu-track.com

构，后面我们会看到它的定义，但是这里从代码中已可看到这个结构中有个 drives[]数组，对应着可以连接到同一 IDE 接口上的同种设备，每个这样的设备都有个读写请求队列。这是一个 request_queue_t 数据结构，前面我们已经列出了它的定义。

找到了具体的队列以后，就要通过由这个队列提供的 make_request_fn 操作创建一个读写请求数据结构，并把它挂入该队列。在某些情况下，有些逻辑上存在的块设备可能并没有直接的对应物，而只是间接地映射到其它块设备上。这种仅在逻辑上存在的块设备只是一种中间的过渡层，因此要提供一个函数来实现相应的映射或完成对最终设备的操作，并在初始化阶段将其操作请求队列结构中的函数指针 make_request_fn 指向这个函数。如果这个函数所完成的只是从一种设备到另一种设备的映射，就返回一个正整数，使 924～935 行的 do-while 循环再执行一遍。而如果完成的是对终极设备的操作，则返回 0，从而结束该 do-while 循环。举例来说，在有容错要求的系统中可以在逻辑上设立一个“可靠硬盘”，并且将文件系统建立在可靠硬盘上，而实际上则把数据平行地存储在两个硬盘上。此时就可以编写一个专用于这种可靠硬盘的驱动程序，并使该设备的操作队列结构中的函数指针 make_request_fn 指向这个函数。当提交一个读写请求时，首先使 buffer_head 结构中的 b_rdev 字段指向第一个硬盘，找到第一个硬盘的读写请求队列（见 925 行，注意这里用的是 b_rdev 中的设备号），并在“可靠硬盘”的 make_request_fn 中为第一个硬盘创建一个读写请求，然后将 buffer_head 结构中的 b_rdev 字段改成指向第二个硬盘，并返回 1。这样，就会再执行一遍 924～935 行的 do-while 循环，为第二个硬盘也创建一个读写请求，但是这一次 make_request_fn 操作返回 0，从而结束这个 do-while 循环。于是，对虚拟的“可靠硬盘”的读写请求就转化成了对两个具体硬盘的读写请求。至于我们在本节所关心的 IDE 硬盘，它本来就已经是终极设备，所以这个函数返回 0。对于普通的 IDE 硬盘，这个函数__make_request()是完成提交操作请求的主体，适用于常规的（终极）块设备。它的主要任务就是为数据缓冲区准备一个操作请求，即 request 数据结构，并将这个数据结构挂入设备的操作请求队列中。如果在此之前该队列是空的，就要启动设备的输入／输出操作。一旦启动以后，对整个队列的操作就是由中断驱动的了。这个由中断驱动的过程一直要到队列中再没有等待着操作的请求时才结束。如果在将操作请求挂入队列之前队列中非空，那就说明由中断驱动的整个过程已经启动，从概念上说只要把新的请求挂在队列的尾部就可以了。但是，实际上常常要进行某种程度的优化。这个函数所作的优化之一，是在队列中由后向前搜索，试图将新的请求与已经在队列中的其它请求合并。合并的条件主要是操作相同（同为读或写）并且扇区连续。有时候，两个请求的扇区相连续，但是缓冲区却不连续（分属不同的页面），此时两个请求仍能合并，但是形成两个不连续的缓冲区“分段”（segment），这是允许的。不过，对于每个 request 结构中可以容纳的分段数量有个限制。在 request 结构中有个 buffer_head 结构的队列，所谓合并就是将多个 buffer_head 结构连在一起成为一个操作请求。由于这个函数的代码较长，我们分段阅读，其代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 695 737 ====================
[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()]

```
695  static int __make_request(request_queue_t * q, int rw,
696                          struct buffer_head * bh)
697  {
698      unsigned int sector, count;
699      int max_segments = MAX_SEGMENTS;
700      struct request * req = NULL, *freereq = NULL;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int rw_ahead, max_sectors, el_ret;
    struct list_head *head;
    int latency;
    elevator_t *elevator = &q->elevator;

    count = bh->b_size >> 9;
    sector = bh->b_rsector;

    rw_ahead = 0; /* normal case; gets changed below for READA */
    switch (rw) {
        case READA:
            rw_ahead = 1;
            rw = READ;    /* drop into READ */
        case READ:
        case WRITE:
            break;
        default:
            BUG();
            goto end_io;
    }

    /* We'd better have a real physical mapping!
       Check this bit only if the buffer was dirty and just locked
       down by us so at this point flushpage will block and
       won't clear the mapped bit under us. */
    if (!buffer_mapped(bh))
        BUG();

    /*
     * Temporary solution - in 2.5 this will be done by the lowlevel
     * driver. Create a bounce buffer if the buffer data points into
     * high memory - keep the original buffer otherwise.
     */
#if CONFIG_HIGHMEM
    bh = create_bounce(rw, bh);
#endif

```

所请求的操作种类可以是 READ、WRITE 以及 READA 三者之一。如果是 READA，即预读，则将预读标志 rw_ahead 设置成 1，并将操作种类改成 READ。记录块的缓冲区必须属于某个缓冲页面，其 buffer_head 结构中的 Mapped 标志位应该为 1，所以这里通过 buffer_mapped()加以确认。如果 CPU 支持超过 32 位（4GB）内存空间（HIGHMEM），而缓冲区又在高于 4GB 的空间中，就要通过 create_bounce()为其在 4GB 以内建立一个镜像，而具体的读写将针对这个镜像进行，完成以后再把其内容复制到高于 4GB 的空间中。这是因为目前 DMA 控制器的寻址能力都不超过 32 位。

继续往下看。

==================== drivers/block/ll_rw_blk.c 738 764 ====================
[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
738 /* look for a free request. */
739      /*
740       * Try to coalesce the new request with old requests
741       */
742      max_sectors = get_max_sectors(bh->b_rdev);
743
744      latency = elevator_request_latency(elevator, rw);
745
746      /*
747       * Now we acquire the request spinlock, we have to be mega careful
748       * not to schedule or do something nonatomic
749       */
750 again:
751      spin_lock_irq(&io_request_lock);
752
753      /*
754       * skip first entry, for devices with active queue head
755       */
756      head = &q->queue_head;
757      if (q->head_active && !q->plugged)
758           head = head->next;
759
760      if (list_empty(head)) {
761           q->plug_device_fn(q, bh->b_rdev); /* is atomic */
762           goto get_rq;
763      }
764
```

这里调用了一个函数 elevator_request_latency()计算出一个数值 latency，那是为后面的另一种优化进行一些准备，我们暂时放一下。我们也暂时跳过这里的 757～758 行。

如果队列原来是空的，那就有个启动 I/O 操作的问题。当然，I/O 的启动要到将第一个请求挂入队列中以后才进行。可是由谁进行呢？一个选择是由当前进程自己直接调用一个函数来启动。另一个选择是将 I/O 的启动作为一个 bottom_half 函数（见第 3 章）“插入”内核中的一个任务队列 tq_disk。这样，每当执行这个任务队列时，就会依次执行队列中等待执行的所有 bottom_half 函数，包括具体块设备对 I/O 的启动。这里（761 行）通过函数指针 plug_device_fn 调用一个由具体队列提供的函数，这个函数可以决定是否将队列插入 tq_disk。就 IDE 硬盘而言，这个指针在初始化时设置成指向 generic_plug_device()，它将设备的操作请求队列插入 tq_disk 中。代表着操作请求队列的 request_queue_t 数据结构中有一个成分 plug_tq，是一个 tq_struct 结构，就是为链入 tq_disk 队列而设的。还有一个成分 plugged，是一个标志，当这个标志为 1 时就表示该数据结构已经在 tq_disk 队列中。当执行任务队列 tq_disk 时，会通过队列中每个 tq_struct 结构里的函数指针 routine 执行一个 bottom-half 函数，启动具体硬盘的 I/O。将操作请求队列插入 tq_disk 以后，就直接转到 get_rq 标号处，进一步处理将操作请求加入请求队列的事。但是，如果操作请求队列原来不是空的话，那就要先试试是否能将新的请求与队列中原有的请求合在一起进行一些优化。

继续往下看__make_request()的代码，这一段就是对优化的尝试。

==================== drivers/block/ll_rw_blk.c 765 803 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()]

```
    el_ret = elevator->elevator_merge_fn(q, &req, bh, rw,
                                         &max_sectors, &max_segments);
    switch (el_ret) {

        case ELEVATOR_BACK_MERGE:
            if (!q->back_merge_fn(q, req, bh, max_segments))
                break;
            req->bhtail->b_reqnext = bh;
            req->bhtail = bh;
            req->nr_sectors = req->hard_nr_sectors += count;
            req->e = elevator;
            drive_stat_acct(req->rq_dev, req->cmd, count, 0);
            attempt_back_merge(q, req, max_sectors, max_segments);
            goto out;

        case ELEVATOR_FRONT_MERGE:
            if (!q->front_merge_fn(q, req, bh, max_segments))
                break;
            bh->b_reqnext = req->bh;
            req->bh = bh;
            req->buffer = bh->b_data;
            req->current_nr_sectors = count;
            req->sector = req->hard_sector = sector;
            req->nr_sectors = req->hard_nr_sectors += count;
            req->e = elevator;
            drive_stat_acct(req->rq_dev, req->cmd, count, 0);
            attempt_front_merge(q, head, req, max_sectors, max_segments);
            goto out;
        /*
         * elevator says don't/can't merge. get new request
         */
        case ELEVATOR_NO_MERGE:
            break;

        default:
            printk("elevator returned crap (%d)\n", el_ret);
            BUG();
    }

```

当一个 IDE 设备的队列中已经有操作请求在等待，而又有新的操作请求到来时，可以作两种不同的优化。一种是因扇区连续而引起的操作合并，另一种是对操作路线所作的优化或者说调度。为此，在 request_queue_t 结构内部设立了一个 elevator_t 数据结构，名为 elevator，此命名从字义上就表明了对磁盘操作的调度类似于对电梯的调度。这种数据结构的定义在 include/linux/elevator.h 和 include/linux/blkdev.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/elevator.h 15 31 ====================

```
/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests when that is implemented. In
 * paging, 'bh' is NULL, and the semaphore is used to wait
 * for read/write completion.
 */
struct request {
	struct list_head queue;
	int elevator_sequence;
	struct list_head table;

	struct list_head *free_list;

	volatile int rq_status;	/* should split this into a few status bits */
#define RQ_INACTIVE		(-1)
#define RQ_ACTIVE		1
#define RQ_SCSI_BUSY		0xffff
```

include/linux/blkdev.h

```
typedef struct elevator_s elevator_t;
```

这个数据结构中提供了几个函数指针用于操作的优化，所以特定的 elevator_t 决定了具体的优化算法。目前的 Linux 内核只提供一种（也许应该说一套）优化算法，那就是 ELEVATOR_LINUS，定义于 include/linux/elevator.h：

==================== include/linux/elevator.h 99 113 ====================

```
#define ELEVATOR_LINUS						\
((elevator_t) {							\
	0,			/* not used */			\
								\
	1000000,			/* read passovers */	\
	2000000,			/* write passovers */	\
	0,			/* max_bomb_segments */\
								\
	0,			/* not used */			\
	0,			/* not used */			\
								\
	elevator_linus,			/* elevator_fn */	\
	elevator_linus_merge,		/* elevator_merge_fn */ \
	elevator_noop_dequeue,		/* dequeue_fn */	\
	})
```

对块设备的数据结构初始化时，将其 request_queue_t 结构内的 elevator 设置成 ELEVATOR_LINUS，所以 765 行实际调用的是 elevator_linus_merge()，其代码在 drivers/block/elevator.c 中：

==================== drivers/block/elevator.c 53 98 ====================
[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()
>elevator_linus_merge()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int elevator_linus_merge(request_queue_t *q, struct request **req,
                struct buffer_head *bh, int rw,
                int *max_sectors, int *max_segments)
{
     struct list_head *entry, *head = &q->queue_head;
     unsigned int count = bh->b_size >> 9, ret = ELEVATOR_NO_MERGE;

     entry = head;
     if (q->head_active && !q->plugged)
          head = head->next;

     while ((entry = entry->prev) != head) {
          struct request *__rq = *req = blkdev_entry_to_request(entry);
          if (__rq->sem)
               continue;
          if (__rq->cmd != rw)
               continue;
          if (__rq->nr_sectors + count > *max_sectors)
               continue;
          if (__rq->rq_dev != bh->b_rdev)
               continue;
          if (__rq->sector + __rq->nr_sectors == bh->b_rsector) {
               ret = ELEVATOR_BACK_MERGE;
               break;
          }
          if (!__rq->elevator_sequence)
               break;
          if (__rq->sector - count == bh->b_rsector) {
               __rq->elevator_sequence--;
               ret = ELEVATOR_FRONT_MERGE;
               break;
          }
     }

     /*
      * second pass scan of requests that got passed over, if any
      */
     if (ret != ELEVATOR_NO_MERGE && *req) {
          while ((entry = entry->next) != &q->queue_head) {
               struct request *tmp = blkdev_entry_to_request(entry);
               tmp->elevator_sequence--;
          }
     }

     return ret;
}
```

这里的 blkdev_entry_to_request()是个宏操作，从一个 list_head 结构找到其宿主 request 结构，定义于 include/linux/blkdev.h：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/blkdev.h 188 188 ====================

```
#define blkdev_entry_to_request(entry) list_entry((entry), struct request, queue)
```

这个函数分两趟扫描当前的读写请求队列，返回对新来到的操作请求应如何与原有操作请求合并的指导（向前、向后、或不能合并），并通过参数 req 返回可以与之合并的操作请求。为不至于过于冲淡操作的主线索，我们把这部分优化的详情留给有兴趣的读者。不过要指出，与新来到的操作请求合并以后可能会为进一步的合并创造条件，所以前面代码中的 777 和 791 行还要调用 attempt_back_merge()或 attempt_front_merge()作进一步的尝试。

如果合并成功，就不需要为新的读写请求单独创建一个数据结构并挂入队列了，否则就要继续往下跑，这就到了标号 get_rq 处。

==================== drivers/block/ll_rw_blk.c 804 848 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()]

```
    /*
     * Grab a free request from the freelist. Read first try their
     * own queue - if that is empty, we steal from the write list.
     * Writes must block if the write list is empty, and read aheads
     * are not crucial.
     */
get_rq:
    if (freereq) {
        req = freereq;
        freereq = NULL;
    } else if ((req = get_request(q, rw)) == NULL) {
        spin_unlock_irq(&io_request_lock);
        if (rw_ahead)
            goto end_io;

        freereq = __get_request_wait(q, rw);
        goto again;
    }

/* fill up the request-info, and add it to the queue */
    req->cmd = rw;
    req->errors = 0;
    req->hard_sector = req->sector = sector;
    req->hard_nr_sectors = req->nr_sectors = count;
    req->current_nr_sectors = count;
    req->nr_segments = 1; /* Always 1 for a new request. */
    req->nr_hw_segments = 1; /* Always 1 for a new request. */
    req->buffer = bh->b_data;
    req->sem = NULL;
    req->bh = bh;
    req->bhtail = bh;
    req->rq_dev = bh->b_rdev;
    req->e = elevator;
    add_request(q, req, head, latency);
out:
```

```
    if (!q->plugged)
        (q->request_fn)(q);
    if (freereq)
        blkdev_release_request(freereq);
    spin_unlock_irq(&io_request_lock);
    return 0;
end_io:
    bh->b_end_io(bh, test_bit(BH_Uptodate, &bh->b_state));
    return 0;
}
```

首先是分配一个 request 结构，这种数据结构是在 include/linux/blkdev.h 中定义的：

==================== include/linux/blkdev.h 15 51 ====================

```
/*
 * Ok, this is an expanded form so that we can use the same
 * request for paging requests when that is implemented. In
 * paging, 'bh' is NULL, and the semaphore is used to wait
 * for read/write completion.
 */
struct request {
    struct list_head queue;
    int elevator_sequence;
    struct list_head table;

    struct list_head *free_list;

    volatile int rq_status;/* should split this into a few status bits */
#define RQ_INACTIVE         (-1)
#define RQ_ACTIVE       1
#define RQ_SCSI_BUSY        0xffff
#define RQ_SCSI_DONE        0xfffe
#define RQ_SCSI_DISCONNECTING    0xffe0

    kdev_t rq_dev;
    int cmd;      /* READ or WRITE */
    int errors;
    unsigned long sector;
    unsigned long nr_sectors;
    unsigned long hard_sector, hard_nr_sectors;
    unsigned int nr_segments;
    unsigned int nr_hw_segments;
    unsigned long current_nr_sectors;
    void * special;
    char * buffer;
    struct semaphore * sem;
    struct buffer_head * bh;
    struct buffer_head * bhtail;
    request_queue_t *q;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        elevator_t *e;
};
```

代码的作者在注释中说以后在页面换入 / 换出机制中也将使用这个数据结构。

内核中可供使用的 request 数据结构的数量是固定的，同时在调用 get_request()时也说明了具体的操作（写操作比读操作慢，所以限制也更严）。如果分配失败，就说明系统中未完成的操作请求已经太多了。怎么办呢？那就要看具体情况了。如果所要求的只不过是预读，那本来就是可做可不做的，所以干脆就无功而返了。否则，要是非做不可的话，那就只好通过__get_request_wait()睡眠等待了。在这个函数中还要针对任务队列 tq_disk 调用 run_task_queue()，使所有可能尚未启动的操作请求队列得到启动，为 request 结构的回收创造条件。当 request 结构的回收使分配这种结构的要求得到满足时，睡眠中的进程就被唤醒而从__get_request_wait()返回。显然，返回时指针 req 必定指向刚分配到的 request 结构而无需再加检验。不过。经过一段时间的睡眠，操作请求队列的情况可能已经变了（例如队列中的若干操作请求可能已经完成而不再在队列中了），所以要回到前面的标号 again 处，重新使指针 head 指向队列中的第一个请求。通常，当开始执行一个操作请求时要将其 request 结构从队列前端摘下，但是有些设备的驱动程序要到操作完成时才将它摘下，所以对这样的操作要跳过队列中的第一个 request 结构，而使指针 head 指向队列中的第二项。为了区分这两种不同的方式，在 request_queue_t 中设立了一个 head_active 字段，表示队列中的第一个 request 结构已经 active，因而应该把它跳过。不过，这只有在操作请求队列已经启动（不再在 tq_disk 队列中）时才需要（见 756～758 行）。当第二次到达 811 行时，freereq 一定已经指向一个空闲的 request 数据结构。

完成了对 request 结构的设置以后，就调用 add_request()将其挂入设备的操作请求队列。这里的参数 latency 是在前面 744 行设置好了的，反映了当时对操作等待时间的预估值，用于优化目的，其实际上来自设备的 elevator_t 数据结构（request_queue_t 结构中的一部分）。

==================== include/linux/elevator.h 72 81 ====================

```
static inline int elevator_request_latency(elevator_t * elevator, int rw)
{
        int latency;

        latency = elevator->read_latency;
        if (rw != READ)
                latency = elevator->write_latency;

        return latency;
}
```

函数 add_request()的代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 582 618 ====================

[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()>add_request()]

```
/*
 * add-request adds a request to the linked list.
 * It disables interrupts (acquires the request spinlock) so that it can muck
 * with the request-lists in peace. Thus it should be called with no spinlocks
 * held.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *
 * By this point, req->cmd is always either READ/WRITE, never READA,
 * which is important for drive_stat_acct() above.
 */

static inline void add_request(request_queue_t * q, struct request * req,
                               struct list_head *head, int lat)
{
     int major;

     drive_stat_acct(req->rq_dev, req->cmd, req->nr_sectors, 1);

     /*
      * let selected elevator insert the request
      */
     q->elevator.elevator_fn(req, &q->elevator, &q->queue_head, head, lat);

       /*
      * FIXME(eric) I don't understand why there is a need for this
      * special case code.  It clearly doesn't fit any more with
      * the new queueing architecture, and it got added in 2.3.10.
      * I am leaving this in here until I hear back from the COMPAQ
      * people.
        */
     major = MAJOR(req->rq_dev);
     if (major >= COMPAQ_SMART2_MAJOR+0 && major <= COMPAQ_SMART2_MAJOR+7)
          (q->request_fn)(q);
     if (major >= COMPAQ_CISS_MAJOR+0 && major <= COMPAQ_CISS_MAJOR+7)
              (q->request_fn)(q);
     if (major >= DAC960_MAJOR+0 && major <= DAC960_MAJOR+7)
          (q->request_fn)(q);
}
```

这里的 drive_stat_acct()只是用来积累一些统计信息，我们不感兴趣。

如果队列原来是空的，那么很简单，通过 list_add()把 request 数据结构挂入队列就是了，request 结构中的 list_head 结构就是为此目的而设的。如果队列中原来就有操作请求呢？显然，这意味着新的请求不能与已经存在的请求合并，所以应将其作为一个单独的请求挂入队列。如果不考虑进一步优化，把新的请求挂在队列的尾端也就可以了。但是，这里考虑了进一步的优化，即对操作路线的优化。所以通过 elevator 数据结构中的函数指针 elevator_fn 调用提供具体优化算法的函数，由这个函数来决定怎样将新的操作请求插入队列中。我们在前面已经看到其 elevator_fn 指针指向 elevator_linus()，其代码在 drivers/block/elevator.c 中：

==================== drivers/block/elevator.c 29 51 ====================
[sys_read()>block_read()>ll_rw_block()>submit_bh()>generic_make_request()>__make_request()>add_request()>elevator_linus()]

```
/*
 * Order ascending, but only allow a request to be skipped a certain
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * number of times
 */
void elevator_linus(struct request *req, elevator_t *elevator,
             struct list_head *real_head,
             struct list_head *head, int orig_latency)
{
     struct list_head *entry = real_head;
     struct request *tmp;

     req->elevator_sequence = orig_latency;

     while ((entry = entry->prev) != head) {
          tmp = blkdev_entry_to_request(entry);
          if (IN_ORDER(tmp, req))
               break;
          if (!tmp->elevator_sequence)
               break;
          tmp->elevator_sequence--;
     }
     list_add(&req->queue, entry);
}
```

这个函数所实现是对磁头移动路线的优化。设想在队列中已经存在在两个操作请求，第一个是对第 100 号柱面上某几个扇区的操作，第二个是对第 800 号柱面上某几个扇区的操作。而新的请求是对第 150 号柱面上某几个扇区的操作。如果简单地把新的请求排在最后，那么在执行时先要把磁头移到第 100 号柱面，完成后再移到第 800 号柱面，最后又要回到第 150 号柱面。磁头的移动定位又是磁盘访问中最费时间的动作。这样的情况一多，磁头就疲于奔命而大大降低了效率。如果新的请求恰巧也在第 100 号柱面而本来可以不必移动磁头，那就更是冤枉了。可是，如果把新的请求插入到第 2 个请求之前，让磁头在完成了第一个请求的操作以后“顺路”先对第 150 号柱面操作，然后再到第 800 号柱面，那就可以显著改善效率了。所以，表面上是选择新请求在队列中的插入点，实质上却是对磁头移动的一种调度。代码中从队列的尾端开始向前扫描，以扇区号为依据（扇区号与柱面号是可以换算的），试图找到这么一个请求，即它所操作的扇区在新请求所操作的扇区之前，因而二者是“顺序”的。如果每次加入一个新操作请求时都保持顺序，那么整个队列就都是顺序的。宏操作 IN_ORDER 的定义在 include/linux/elevator.h 中：

==================== include/linux/elevator.h 61 70 ====================

```
/*
 * This is used in the elevator algorithm.  We don't prioritise reads
 * over writes any more --- although reads are more time-critical than
 * writes, by treating them equally we increase filesystem throughput.
 * This turns out to give better overall performance.  -- sct
 */
#define IN_ORDER(s1,s2)                 \
     ((((s1)->rq_dev == (s2)->rq_dev &&    \
        (s1)->sector < (s2)->sector)) ||   \
      (s1)->rq_dev < (s2)->rq_dev)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里把两个请求不在同一设备上、但是前者的次设备号小于后者的次设备号也考虑作顺序。但是，对于 IDE 接口，是每个设备一个队列（见 ide_get_queue()的代码），所以这一点对 IDE 设备实际上不起作用。

如果找到了符合要求的操作请求，就把新请求插入到它的后面，从而实现了某种程度的优化。如果找不到，就把新请求挂在队列的尾端，也就是说无法优化了。

我们在这里不对这些优化作定量的分析。一个优化算法应满足两个基本要求。一是有效性，即在多数可以优化的场合下真的得到了优化，提高了效率。二是公正性。以合并为例，如果我们每次都从队列的前端开始扫描，那么就有可能发生这样的情况：队列中原来就已经有很多操作请求在等待了，突然来了一大批新的操作请求，这些请求恰好都可以跟处于队列前端的请求合并，于是就全部“加塞 ”插到了大量原已在队列中等待的请求之前。这对于那些已经在等待的请求来说就不公平了。在极端的情况下，这些请求甚至可能“饿死”（starving）而一直得不到服务。作为队列，基本上的“先来先服务 ”原则还是应该遵循的。这就是为什么在优化和合并时都从队列尾端开始向前扫描的原因，并且即使从队列尾端开始向前扫描也还应该对扫描的距离加以限制。最后，还有代价的考虑，优化的问题常常是 NP 完全问题，所以在实际运用时都要“适而可止”，不能追求完美。

至此，将新的请求挂入操作请求队列的任务已经完成了，在 add_request()中，还要检查一下操作的对象是否若干种设备之一（611～617 行），如果是就要直接通过队列的函数指针 request_fn 启动队列中的第一个 I/O 操作。回到__make_request()中，剩下的事情只有一件，那就是：如果操作请求队列的 plugged 标志为 0，也就是说尚未把队列插入到 tq_disk 中，此时就要直接通过队列的函数指针 request_fn 启动队列中的第一个 I/O 操作。正因为这样，代码的作者在__make_request()中加了注释，说既然有了后者就没有必要在 add_request()中考虑那些特殊情况了。不过，在我们所讲的情景中，我们假定在__make_request()中已经将它插入 bottom-half 的任务队列 tq_disk，所以 plugged 标志为 1。

这样，函数 ll_rw_block()的执行就完成了。注意此时所要求的读 / 写实际上显然并未完成。实际的读 / 写是一个异步的过程，我们无法预测它究竟会在何时发生，因此需要使用或确认读 / 写结果的进程必须等待以取得同步。这通常是通过 wait_on_buffer()完成的。这是一个 inline 函数，其代码在 include/linux/locks.h 中：

==================== include/linux/locks.h 17 21 ====================

```
extern inline void wait_on_buffer(struct buffer_head * bh)
{
	if (test_bit(BH_Lock, &bh->b_state))
		__wait_on_buffer(bh);
}
```

函数__wait_on_buffer()的代码在 fs/buffer.c 中：

==================== fs/buffer.c 136 162 ====================

```
/*
 * Rewrote the wait-routines to use the "new" wait-queue functionality,
 * and getting rid of the cli-sti pairs. The wait-queue routines still
 * need cli-sti, but now it's just a couple of 386 instructions or so.
 *
 * Note that the real wait_on_buffer() is an inline function that checks
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * if 'b_wait' is set before calling this, so that the queues aren't set
 * up unnecessarily.
 */
void __wait_on_buffer(struct buffer_head * bh)
{
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);

    atomic_inc(&bh->b_count);
    add_wait_queue(&bh->b_wait, &wait);
    do {
        run_task_queue(&tq_disk);
        set_task_state(tsk, TASK_UNINTERRUPTIBLE);
        if (!buffer_locked(bh))
            break;
        schedule();
    } while (buffer_locked(bh));
    tsk->state = TASK_RUNNING;
    remove_wait_queue(&bh->b_wait, &wait);
    atomic_dec(&bh->b_count);
}
```

阅读了本书第 4 章和第 3 章的读者，对这段代码应该不会感到困难。等待着使用指定缓冲区（的内容）的进程在一个等待队列 wait 中睡眠，等待操作的完成。而对给定缓冲区的操作是否已经完成的标志，就是为了对缓冲区的操作所加的锁是否已被解除。所以，当给定缓冲区的 I/O 操作完成时，应该为该缓冲区解锁，并唤醒在该队列中睡眠的进程。但是，被唤醒并不一定意味着对特定缓冲区的操作已经完成，所以要在一个 do-while 循环中反复地测试和进入睡眠。另一方面，进程每次在 do-while 循环中进入睡眠前要通过 run_task_queue()执行在 tq_disk 队列中的 bottom_half 函数，以确保对操作请求队列的执行已经启动。不过，对 run_task_queue()的调用实际上可以来自好多不同的函数和场合，这里的调用并不是惟一的。

不管是从什么途径，当对 tq_disk 队列调用 run_task_queue()时，就会依次把队列中的 tq_struct 数据结构从队列中解除并通过其函数指针 routine 调用相应的 bottom_half 函数。前面讲过，对于块设备操作请求队列，这个函数是 generic_unplug_device()，现在我们来看它的代码，这是在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 367 375 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()]

```
static void generic_unplug_device(void *data)
{
    request_queue_t *q = (request_queue_t *) data;
    unsigned long flags;

    spin_lock_irqsave(&io_request_lock, flags);
    __generic_unplug_device(q);
    spin_unlock_irqrestore(&io_request_lock, flags);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/block/ll_rw_blk.c 355 365 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()]

```
/*
 * remove the plug and let it rip..
 */
static inline void __generic_unplug_device(request_queue_t *q)
{
	if (q->plugged) {
		q->plugged = 0;
		if (!list_empty(&q->queue_head))
			q->request_fn(q);
	}
}
```

在 tq_struct 结构中有一个 void 指针 data，用来传递对 bottom-half 函数的参数。对于操作请求队列结构中的 tq_struct 结构 plug_tq，这个指针就指向其宿主 request_queue 数据结构。只要这个队列非空，这里就通过其函数指针 request_fn 调用该队列的 I/O 启动函数。这个启动函数是在块设备初始化时设置好的，对于作为主设备（primary）连接在 IDE 接口上的 IDE 硬盘，它是 do_ide_request()，其代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 1377 1383 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()]

```
/*
 * Passes the stuff to ide_do_request
 */
void do_ide_request(request_queue_t *q)
{
	ide_do_request(q->queuedata, 0);
}
```

操作请求队列中，即 request_queue_t 结构中，有个 void 指针 queuedata，可以根据具体设备的不同而指向不同的对象。对于 IDE 接口，这个指针指向一个 ide_hwgroup_t 数据结构，这也是在 include/linux/ide.h 中定义的：

==================== include/linux/ide.h 486 503 ====================

```
/*
 * when ide_timer_expiry fires, invoke a handler of this type
 * to decide what to do.
 */
typedef int (ide_expiry_t)(ide_drive_t *);

typedef struct hwgroup_s {
	ide_handler_t		*handler;/* irq handler, if active */
	volatile int		busy;	/* BOOL: protects all fields below */
	int		sleeping; /* BOOL: wake us up on timer expiry */
	ide_drive_t		*drive;	/* current drive */
	ide_hwif_t		*hwif;	/* ptr to current hwif in linked-list */
	struct request		*rq; /* current request */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    struct timer_list  timer;   /* failsafe timer */
    struct request         wrq; /* local copy of current write rq */
    unsigned long      poll_timeout; /* timeout value during long polls */
    ide_expiry_t       *expiry; /* queried upon timeouts */
} ide_hwgroup_t;
```

这个数据结构代表着一个“硬件组”，实际上是一组 IDE 接口，对属于同一组的 IDE 接口（以及接口上的磁盘）不能同时操作。不过，在大部分情况下每个 IDE 接口都能独立操作而互不影响，因而所谓一组 IDE 接口只包括一个 IDE 接口。IDE 接口是由 ide_hwif_t 数据结构代表的，内核中有个 ide_hwif_t 数组 ide_hwifs[]，以 IDE 接口的编号为下标。

==================== drivers/ide/ide.c 191 191 ====================

```
ide_hwif_t    ide_hwifs[MAX_HWIFS];  /* master data repository */
```

这里的 MAX_HWIFS 定义为 6。一般的 PC 机有两个 IDE 硬盘接口，各自支持最多两个硬盘，此外还有一个 IDE 软盘接口，再要增加就得外加 IDE 接口了。数组 ide_hwifs[]中的每个元素都是一个 ide_hwif_t 数据结构，代表着系统中的一个 IDE 接口，定义于 include/linux/ide.h 中：

==================== include/linux/ide.h 421 470 ====================

```
typedef struct hwif_s {
    struct hwif_s *next;        /* for linked-list in ide_hwgroup_t */
    void      *hwgroup;/* actually (ide_hwgroup_t *) */
    ide_ioreg_t   io_ports[IDE_NR_PORTS];/* task file registers */
    hw_regs_t hw;       /* Hardware info */
    ide_drive_t   drives[MAX_DRIVES];     /* drive info */
    struct gendisk     *gd;      /* gendisk structure */
    ide_tuneproc_t     *tuneproc;     /* routine to tune PIO mode for drives */
    ide_speedproc_t    *speedproc;    /* routine to retune DMA modes for drives */
    ide_selectproc_t *selectproc;     /* tweaks hardware to select drive */
    ide_resetproc_t    *resetproc;    /* routine to reset controller after a disk reset */
    ide_intrproc_t     *intrproc;     /* special interrupt handling for shared pci interrupts */
    ide_maskproc_t     *maskproc;     /* special host masking for drive selection */
    ide_quirkproc_t    *quirkproc;    /* check host's drive quirk list */
    ide_rw_proc_t *rwproc;  /* adjust timing based upon rq->cmd direction */
    ide_dmaproc_t *dmaproc;/* dma read/write/abort routine */
    unsigned int  *dmatable_cpu;     /* dma physical region descriptor table (cpu view) */
    dma_addr_t    dmatable_dma; /* dma physical region descriptor table (dma view) */
    struct scatterlist *sg_table;    /* Scatter-gather list used to build the above */
    int sg_nents;           /* Current number of entries in it */
    int sg_dma_direction;        /* dma transfer direction */
    struct hwif_s *mate;         /* other hwif from same PCI chip */
    unsigned long dma_base;/* base addr for dma ports */
    unsigned dma_extra;     /* extra addr for dma ports */
    unsigned long config_data;  /* for use by chipset-specific code */
    unsigned long select_data;  /* for use by chipset-specific code */
    struct proc_dir_entry *proc;     /* /proc/ide/ directory entry */
    int      irq;      /* our irq number */
    byte     major;         /* our major number */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        char            name[6]; /* name of interface, eg. "ide0" */
        byte        index;          /* 0 for ide0; 1 for ide1; ... */
        hwif_chipset_t      chipset; /* sub-module for tuning.. */
        unsigned noprobe    : 1;     /* don't probe for this interface */
        unsigned present    : 1;     /* this interface exists */
        unsigned serialized : 1;     /* serialized operation with mate hwif */
        unsigned sharing_irq: 1;     /* 1 = sharing irq with another hwif */
        unsigned reset      : 1;     /* reset after probe */
        unsigned autodma    : 1;     /* automatically try to enable DMA at boot */
        unsigned udma_four  : 1;     /* 1=ATA-66 capable, 0=default */
        byte        channel; /* for dual-port chips: 0=primary, 1=secondary */
#ifdef CONFIG_BLK_DEV_IDEPCI
        struct pci_dev      *pci_dev;/* for pci chipsets */
        ide_pci_devid_t     pci_devid;    /* for pci chipsets: {VID,DID} */
#endif /* CONFIG_BLK_DEV_IDEPCI */
#if (DISK_RECOVERY_TIME > 0)
        unsigned long last_time;     /* time when previous rq was done */
#endif
        byte        straight8;     /* Alan's straight 8 check */
        void        *hwif_data;    /* extra hwif data */
} ide_hwif_t;
```

结构中的数组 drives[]对应着可以连接到同一个 IDE 接口上的若干个硬盘，常数 MAX_DRIVES 定义为 2，所以数组的大小为 2。指针 hwgroup 就指向一个 ide_hwgroup_t 结构。

通常，每个 IDE 接口都有一个 ide_hwgroup_t 数据结构，二者以它们的指针 hwgroup 和 hwif 互相指向对方。同一个 IDE 接口上的两个磁盘不能同时操作，所以 ide_hwgroup_t 结构代表着一组不能同时操作而必须把对它们的操作“串行化”的磁盘。但是，也有些 IDE 接口卡把两个 IDE 接口做在一起而不能同时操作，于是就共用一个 ide_hwgroup_t 数据结构，此时两个 ide_hwif__t 结构就要通过它们的指针 next 链接在一起，而 ide_hwgroup_t 结构中的 hwif 指针则指向当前 IDE 接口。所以 ide_hwgroup_t 是比 ide_hwif_t 更高一层的数据结构。也就是说：

- 每个 ide_hwgroup_t 数据结构代表着一组不能同时操作的 IDE 接口。
- 每个 ide_hwif_t 数据结构代表着一个 IDE 接口，也就是一组不能同时操作的 IDE 设备，ide_hwif_t 结构中有个 ide_drive_t 结构数组。
- 每个 ide_drive_t 数据结构代表着一个 IDE 设备。

在 PC 机中，通常一个“硬件组”只包含一个 IDE 接口，而一个 IDE 接口包含一个或两个磁盘。

每个 IDE 设备，即 ide_drive_t 数据结构，都有自己的操作请求队列，但是整个 IDE 接口组只能有一个“当前操作请求”，即正在为之服务的操作请求，ide_hwgroup_t 结构中的指针 rq 就指向这个请求的 request 数据结构。

回过来看 ide_do_request()的代码，这是在 drivers/ide/ide.c 中，代码的作者在函数前面加了很长一段注释，读者不妨先读一下：

==================== drivers/ide/ide.c 1254 1288 ====================

```
/*
 * Issue a new request to a drive from hwgroup
```

```
 * Caller must have already done spin_lock_irqsave(&io_request_lock, ..);
 *
 * A hwgroup is a serialized group of IDE interfaces.  Usually there is
 * exactly one hwif (interface) per hwgroup, but buggy controllers (eg. CMD640)
 * may have both interfaces in a single hwgroup to "serialize" access.
 * Or possibly multiple ISA interfaces can share a common IRQ by being grouped
 * together into one hwgroup for serialized access.
 *
 * Note also that several hwgroups can end up sharing a single IRQ,
 * possibly along with many other devices.  This is especially common in
 * PCI-based systems with off-board IDE controller cards.
 *
 * The IDE driver uses the single global io_request_lock spinlock to protect
 * access to the request queues, and to protect the hwgroup->busy flag.
 *
 * The first thread into the driver for a particular hwgroup sets the
 * hwgroup->busy flag to indicate that this hwgroup is now active,
 * and then initiates processing of the top request from the request queue.
 *
 * Other threads attempting entry notice the busy setting, and will simply
 * queue their new requests and exit immediately.  Note that hwgroup->busy
 * remains set even when the driver is merely awaiting the next interrupt.
 * Thus, the meaning is "this hwgroup is busy processing a request".
 *
 * When processing of a request completes, the completing thread or IRQ-handler
 * will start the next request from the queue.  If no more work remains,
 * the driver will clear the hwgroup->busy flag and exit.
 *
 * The io_request_lock (spinlock) is used to protect all access to the
 * hwgroup->busy flag, but is otherwise not needed for most processing in
 * the driver.  This makes the driver much more friendlier to shared IRQs
 * than previous designs, while remaining 100% (?) SMP safe and capable.
 */
```

读者可以先大致看一下，然后在阅读有关的代码时再回过来细读。下面是函数的代码：

==================== drivers/ide/ide.c 1289 1365 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()]

```
static void ide_do_request(ide_hwgroup_t *hwgroup, int masked_irq)
{
    ide_drive_t    *drive;
    ide_hwif_t     *hwif;
    ide_startstop_t    startstop;

    ide_get_lock(&ide_lock, ide_intr, hwgroup);    /* for atari only: POSSIBLY BROKEN HERE(?) */

    __cli(); /* necessary paranoia: ensure IRQs are masked on local CPU */

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    while (!hwgroup->busy) {
        hwgroup->busy = 1;
        drive = choose_drive(hwgroup);
        if (drive == NULL) {
            unsigned long sleep = 0;
            hwgroup->rq = NULL;
            drive = hwgroup->drive;
            do {
                if (drive->sleep && (!sleep || 0 < (signed long)(sleep - drive->sleep)))
                    sleep = drive->sleep;
            } while ((drive = drive->next) != hwgroup->drive);
            if (sleep) {
                /*
                 * Take a short snooze, and then wake up this hwgroup again.
                 * This gives other hwgroups on the same a chance to
                 * play fairly with us, just in case there are big differences
                 * in relative throughputs.. don't want to hog the cpu too much.
                 */
                if (0 < (signed long)(jiffies + WAIT_MIN_SLEEP - sleep))
                    sleep = jiffies + WAIT_MIN_SLEEP;
#if 1
                if (timer_pending(&hwgroup->timer))
                    printk("ide_set_handler: timer already active\n");
#endif
                hwgroup->sleeping = 1; /* so that ide_timer_expiry knows what to do */
                mod_timer(&hwgroup->timer, sleep);
                /* we purposely leave hwgroup->busy==1 while sleeping */
            } else {
                /* Ugly, but how can we sleep for the lock otherwise? perhaps from tq_disk? */
                ide_release_lock(&ide_lock);     /* for atari only */
                hwgroup->busy = 0;
            }
            return;       /* no more work for this hwgroup (for now) */
        }
        hwif = HWIF(drive);
        if (hwgroup->hwif->sharing_irq && hwif != hwgroup->hwif && hwif->io_ports[IDE_CONTROL_OFFSET]) {
            /* set nIEN for previous hwif */
            SELECT_INTERRUPT(hwif, drive);
        }
        hwgroup->hwif = hwif;
        hwgroup->drive = drive;
        drive->sleep = 0;
        drive->service_start = jiffies;

        if ( drive->queue.plugged ) /* paranoia */
            printk("%s: Huh? nuking plugged queue\n", drive->name);
        hwgroup->rq = blkdev_entry_next_request(&drive->queue.queue_head);
        /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         * Some systems have trouble with IDE IRQs arriving while
         * the driver is still setting things up.  So, here we disable
         * the IRQ used by this interface while the request is being started.
         * This may look bad at first, but pretty much the same thing
         * happens anyway when any interrupt comes in, IDE or otherwise
         *  -- the kernel masks the IRQ while it is being handled.
         */
        if (masked_irq && hwif->irq != masked_irq)
            disable_irq_nosync(hwif->irq);
        spin_unlock(&io_request_lock);
        ide__sti();   /* allow other IRQs while we start this request */
        startstop = start_request(drive);
        spin_lock_irq(&io_request_lock);
        if (masked_irq && hwif->irq != masked_irq)
            enable_irq(hwif->irq);
        if (startstop == ide_stopped)
            hwgroup->busy = 0;
    }
}
```

对于 i386 结构的 CPU，ide_get_lock()只是一个空函数。同时，对设备的操作必须是独占的，绝不容许受到干扰。所以这里采取了最为严厉的措施，即通过__cli()把中断关掉，这样就既不会有中断发生，也不会有进程调度发生了。那么，在 SMP 多处理器结构的系统中来自其它 CPU 的干扰呢？那也已经考虑到了，请回过去看 generic_unplug_device()的代码，通过函数指针 request_fn 对 do_ide_request()的调用是在由 spin_lock_irqsave()和 spin_unlock_irqrestore()围成的临界区中进行的，一个 CPU 进入了这个区域就把大门锁上了，别的 CPU 想要进来就只好在 spin_lock_irqsave()中等待。

然后，只要 IDE 接口不是正在忙，就可以进入 while 循环的循环体内，并且马上就把这个 IDE 接口的 busy 标志设成 1，又关上了一道门。这道门一关上，即使把外层的大门打开，例如把中断打开，也不会有其它进程（或中断服务程序）能进入这个 while 循环了。不过，现在还不能把中断打开，因为中断服务程序有可能不经由这里的 while 循环而直接访问 IDE 接口造成干扰。

由于同一个 IDE 接口上可能有不止一个磁盘，这时候就要确定接下去到底是对哪一个磁盘操作了。这就是调用 choose_drive()的目的。如前所述，在 ide_hwif_t 数据结构中有个 ide_drive_t 结构数组 drives[]，而相应的 ide_hwgroup_t 结构中则有一个 ide_drive_t 结构指针 drive，它指向上述 drives[]数组中的一个数据结构，这就是整个组的当前磁盘。不过，这个指针所指向的实际上是个环形链，因为存在于同一 IDE 接口上的所有磁盘所对应的 ide_drive_t 数据结构都通过结构中的 next 指针连成一个环形链。所以 choose_drive()的任务就是扫描这个环形链，察看每个 ide_drive_t 结构中的操作请求队列，以确定哪一个磁盘应该是下一步的操作对象。

此处先请读者暂时停一下，先将上面提到的有关 IDE 硬盘驱动的几个数据结构以及相互之间的关系总结一下，并画出联系图。这对程序理解会有好处。

前面说过，同一个“硬件组”，即共用同一个 ide_hwgroup_t 结构的硬盘（通常在同一个 IDE 接口上）是不能同时操作的。但是，这句话不够确切。更确切地应该说是不能同时启动操作，或者说不能同时向两个硬盘发出操作命令，两个磁盘的物理操作（例如移动磁头然后读出）完全可以并行。当向

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

一个硬盘发出操作命令启动其操作的时候，就在相应 ide_drive_t 数据结构里的 service_start 字段中记下当时的时间，根据这个起始时间以及具体操作的大小就可以估计出这个操作完成的时间。另一方面，每当一个操作完成（或到超时而失败）时，可以根据当时的时间和操作的启动时间计算出本次操作实际耗用的时间。这个时间记录在相应 ide_drive_t 结构中的 sevice_time 字段中，可以用作具体硬盘操作速度的一个参考值。在某些场合下，还可以在自愿的基础上（ide_drive_t 结构中的 nice1 标志为 1）将一个硬盘投入睡眠状态，也就是在预定的某个时间之前不对其进行操作，这个预定的时间就记录在 ide_drive_t 结构里的 sleep 字段中。不过要注意，这里所说的是在一定的时间内不对其进行操作，也就是不向其发出操作命令，但是这并不意味着在这段时间内它就一定是在空转，只不过是估计在此之前正在进行中的操作肯定不会完成而已。在挑选下一个操作对象时，这些因素都应该考虑进去。函数 choose_drive()的代码在 drivers/ide/ide.c 中，我们就把它留给读者了。

挑选下一个操作对象的结果有可能是空。原因可能有二，一是所有磁盘的操作请求队列都是空，因而无事可干；二是所有的磁盘都在睡眠状态，不能向它们发出操作命令。所以，代码中通过一个 do-while 循环扫描 ide_drive_t 结构的环形链，找出其中最早的睡眠结束时间。如果这个最早的结束时间非 0，那就说明找不出一个操作对象的原因确是因睡眠引起的，因此将 ide_hwgroup_t 结构中的 sleeping 标志设成 1，表示整个磁盘组都在睡眠（1323 行）；同时设置一个定时器，让它在到点的时候将该 ide_hwgroup_t 结构“唤醒”。否则，如果最早的睡眠结束时间为 0，就说明该组磁盘的操作请求队列全是空的，所以把它的 busy 标志设成 0。既然没有下一个操作对象，本次 ide_do_request()就结束了（见 1331 行）。

如果找到了下一个操作对象（见 1333 行），那就先找到该操作对象，即磁盘所属的 IDE 接口，这里宏操作 HWIF 的定义见 include/linux/ide.h：

==================== include/linux/ide.h 96 96 ====================

```
#define HWIF(drive)         ((ide_hwif_t *)((drive)->hwif))
```

读者也许感到奇怪，在 ide_hwgroup_t 结构中不是有一个指针 hwif 指向当前 ide_hwif_t 数据结构吗？是的，但是如果有两个 IDE 接口共用同一个 ide_hwgroup_t 结构的话，新挑选的 ide_drive_t 结构可能不属于原先的“当前 IDE 接口”。如果真是这样的话（见 1334 行），那就意味着 IDE 接口的切换，这时候就要先向原 IDE 接口的控制寄存器（如果存在的话，见 1334 行）写一个控制字节将该接口的中断请求关闭。这里的宏操作 SELECT_INTERRUPT 定义于 include/linux/ide.h：

==================== include/linux/ide.h 189 195 ====================

```
#define SELECT_INTERRUPT(hwif,drive)                \
{                                                   \
    if (hwif->intrproc)                             \
        hwif->intrproc(drive);                      \
    else                                            \
        OUT_BYTE((drive)->ctl|2, hwif->io_ports[IDE_CONTROL_OFFSET]);    \
}
```

而宏操作 OUT_BYTE 的第 1 个参数为需要写出的字节，当这个字节 bit1 为 1 时，该 IDE 接口的中断请求就被屏蔽掉了。第 2 个参数为寄存器的 I/O 地址，代表着具体 IDE 接口的 ide_hwif_t 结构中有个数组 io_ports[]，该数组提供了接口上各个寄存器的 I/O 地址。

然后，就可以使 ide_hwgroup_t 结构中的指针 hwif 和 drive 指向新的当前 IDE 接口和磁盘了。全局量 jiffies 代表当前时间（见第 3 章），这就是本次操作的启动时间 sevice_start。代码中还对操作对象的操作请求队列加以检验，确保它已经不在 tq_disk 队列中。

接着，使指针 hwgroup->rq 指向操作请求队列中的第一个操作请求（但不将其从队列中脱链），这就是当前操作请求。

至此，整个系统的中断仍是关着的，但是马上就要把它打开了（见 1357 行），因为为了一个具体的设备而长时间关闭全系统的中断机制是不合适的。可是，对当前磁盘的操作仍不允许打扰，所以，在打开全系统的中断之前可能需要先通过 disable_irq_nosync()将具体的中断服务程序关闭，也就是说在打开大门之前先把里面具体房间的门关上。不过，这也是有条件的，这里 masked_irq 是作为参数从 do_ide_request()传下来的，从它的代码中可以看到这个参数是 0，所以这里 1354 行 if 语句的条件不会得到满足，也就是说不必先把里面具体房间的门关上。如前所述，在我们这条执行路线中实际上已经关上了一道门，那就是 IDE 接口的 busy 标志。

做好了这些准备，现在可以根据具体的操作请求向目标磁盘发出操作命令了，这是由 start_request()完成的，其代码在 drivers/ide/ide.c。读者将会看到，这是一个很重要的函数。

==================== drivers/ide/ide.c 1129 1196 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()>start_request()]

```
/*
 * start_request() initiates handling of a new I/O request
 */
static ide_startstop_t start_request (ide_drive_t *drive)
{
	ide_startstop_t startstop;
	unsigned long block, blockend;
	struct request *rq = blkdev_entry_next_request(&drive->queue.queue_head);
	unsigned int minor = MINOR(rq->rq_dev), unit = minor >> PARTN_BITS;
	ide_hwif_t *hwif = HWIF(drive);

#ifdef DEBUG
	printk("%s: start_request: current=0x%08lx\n", hwif->name, (unsigned long) rq);
#endif
	if (unit >= MAX_DRIVES) {
		printk("%s: bad device number: %s\n", hwif->name, kdevname(rq->rq_dev));
		goto kill_rq;
	}
#ifdef DEBUG
	if (rq->bh && !buffer_locked(rq->bh)) {
		printk("%s: block not locked\n", drive->name);
		goto kill_rq;
	}
#endif
	block    = rq->sector;
	blockend = block + rq->nr_sectors;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1156     if ((rq->cmd == READ || rq->cmd == WRITE) &&
1157         (drive->media == ide_disk || drive->media == ide_floppy)) {
1158         if ((blockend < block) || (blockend > drive->part[minor&PARTN_MASK].nr_sects)) {
1159             printk("%s%c: bad access: block=%ld, count=%ld\n", drive->name,
1160              (minor&PARTN_MASK)?'0'+(minor&PARTN_MASK):' ', block, rq->nr_sectors);
1161             goto kill_rq;
1162         }
1163         block += drive->part[minor&PARTN_MASK].start_sect + drive->sect0;
1164     }
1165     /* Yecch - this will shift the entire interval,
1166        possibly killing some innocent following sector */
1167     if (block == 0 && drive->remap_0_to_1 == 1)
1168         block = 1;  /* redirect MBR access to EZ-Drive partn table */
1169
1170 #if (DISK_RECOVERY_TIME > 0)
1171     while ((read_timer() - hwif->last_time) < DISK_RECOVERY_TIME);
1172 #endif
1173
1174     SELECT_DRIVE(hwif, drive);
1175     if (ide_wait_stat(&startstop, drive, drive->ready_stat, BUSY_STAT|DRQ_STAT, WAIT_READY)) {
1176         printk("%s: drive not ready for command\n", drive->name);
1177         return startstop;
1178     }
1179     if (!drive->special.all) {
1180         if (rq->cmd == IDE_DRIVE_CMD || rq->cmd == IDE_DRIVE_TASK) {
1181             return execute_drive_cmd(drive, rq);
1182         }
1183         if (drive->driver != NULL) {
1184             return (DRIVER(drive)->do_request(drive, rq, block));
1185         }
1186         printk("%s: media type %d not supported\n", drive->name, drive->media);
1187         goto kill_rq;
1188     }
1189     return do_special(drive);
1190 kill_rq:
1191     if (drive->driver != NULL)
1192         DRIVER(drive)->end_request(0, HWGROUP(drive));
1193     else
1194         ide_end_request(0, HWGROUP(drive));
1195     return ide_stopped;
1196 }
```

首先是对参数的检查。这里的局部量 block 和 blockend 代表着操作的起始扇区号和结束扇区号，而不是逻辑块号。值得注意的是 1163 行，如果所操作的“磁盘”实际上只是磁盘上的一个分区，那就要将逻辑磁盘的扇区号映射成物理磁盘上的扇区号。在物理磁盘划分成多个分区的情况下，相应 ide_drive_t 结构中的数组 part[]提供了各个分区的参数，包括各个分区的起始逻辑扇区号。这个扇区号之所以仍旧是“逻辑扇区号”，是因为磁盘上存在着一个“分区表”，这个分区表使逻辑上的扇区 0 移到了另一个物理位置上，这个位置就是 drive->sect0。此外，在有磁盘分区存在的情况下，有可能要将

用于引导块的扇区 0 映射到扇区 1 上，此时 ide_drive_t 结构中的 remap0_to_1 标志为 1（见 1167 行）。这些信息都是在块设备初始化时设置好了的。

有些磁盘在相继的两次操作之间要有个“恢复时间”，对这样的磁盘要通过 1171 行的 while 循环保证其恢复时间，但是大多数磁盘都没有这个要求。

下面的 SELECT_DRIVE 是个宏操作，与 SELECT_INTERRUPT 相似，其定义见 include/linux/ide.h：

==================== include/linux/ide.h 182 187 ====================

```
#define SELECT_DRIVE(hwif,drive)                        \
{                                               \
    if (hwif->selectproc)                       \
        hwif->selectproc(drive);                \
    OUT_BYTE((drive)->select.all, hwif->io_ports[IDE_SELECT_OFFSET]); \
}
```

有些 IDE 设备要求在写入其“驱动器 / 磁头选择寄存器”之前先执行一些特殊的处理，所以这样的设备可以将相应 ide_drive_t 结构中的函数指针 selectproc 预先设置好，这里就可以通过函数指针调用这个函数了。但是一般 IDE 磁盘都没有这个要求。这是主要是设置硬盘的“驱动器 / 磁头选择寄存器”，这个 8 位寄存器的格式如阁 8.6 所示。

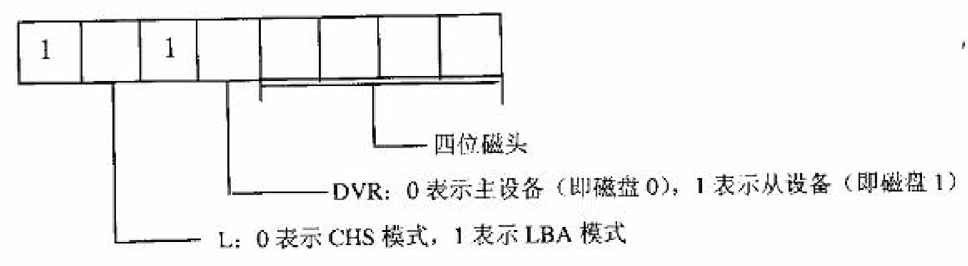


图 8.6　驱动器 / 磁头选择寄存器格式定义

相应的数据结构定义于 include/asm-i386/ide.h 中：

==================== include/asm-i386/ide.h 89 98 ====================

```
typedef union {
    unsigned all            : 8; /* all of the bits together */
    struct {
        unsigned head       : 4; /* always zeros here */
        unsigned unit       : 1; /* drive select number, 0 or 1 */
        unsigned bit5       : 1; /* always 1 */
        unsigned lba        : 1; /* using LBA instead of CHS */
        unsigned bit7       : 1; /* always 1 */
    } b;
    } select_t;
```

对 IDE 磁盘上的扇区有两种不同的寻址方式。第一种是所谓 CHS（柱面 / 磁头 / 扇区）模式。由于“驱动器 / 磁头选择寄存器”中只有 4 位用于磁头选择，最多就只能有 16 个磁头。同时，另外两个用于柱面号的寄存器又将柱面数限制到 1024，而扇区号寄存器又将（每磁道）扇区的数量限制到 256。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

可是，要把一个圆（磁道）划分成 256 个扇区是有困难的，所以实际上只能划分成 64 个扇区。这么一来，扇区的总量就限制为 1024×16×64=1M，由于每个扇区的容量为 512 字节，IDE 磁盘的总容量就被限制为 0.5GB。为了克服这个缺点，在“扩充 IDE”的标准中又提供了一种 LBA（逻辑块号）模式，将整个磁盘看作连续的逻辑扇区阵列，而由 IDE 磁盘（更确切地说是 EIDE 磁盘）自己进行从逻辑扇区号到柱面、磁头、扇区的转换，这样，由原来两个用于柱面号的寄存器，加上原来用于扇区号的寄存器，再加上原来用于磁头号的 4 位，一共有 28 位用于逻辑区号，使理论上的最大容量达到 128 GB。将驱动器 / 磁头选择寄存器中的 bit6 设置成 1，就使磁盘工作于 LBA 模式。

有些 IDE 设备有些特殊操作，例如在每次读 / 写以后都要让磁盘回到 0 号柱面等等。特殊操作带有初始化的性质而只需执行一次。为了照顾到这些有特殊要求的磁盘，ide_drive_t 结构中设置了一个字段 special，它的类型是 special_t，定义见 include/linux/ide.h：

==================== include/linux/ide.h 268 277 ====================

```
typedef union {
	unsigned all		: 8; /* all of the bits together */
	struct {
		unsigned set_geometry	: 1; /* respecify drive geometry */
		unsigned recalibrate	: 1; /* seek to cyl 0	*/
		unsigned set_multmode	: 1; /* set multmode count */
		unsigned set_tune	: 1; /* tune interface for drive */
		unsigned reserved	: 4; /* unused */
	} b;
} special_t;
```

块设备初始化时，如果具体的磁盘有特殊要求，就相应地设置其 ide_drive_t 结构中这个字段的值，否则设置成 0。

先看没有特殊要求的情景（见 1179 行），这是主流。对 IDE 磁盘的操作可以分成两类。一类是起控制作用的命令，如单纯的磁头定位、自检、光盘驱动器的关门 / 开门等。这一类的命令由 execute_drive_cmd()启动，这些操作请求通常不是由用户进程（应用程序）发出的。另一类就是我们所关心的数据读 / 写命令，通过具体设备的驱动函数跳转表，即 ide_driver_t 结构中的函数指针 do_request 启动，宏操作 DRIVER 的定义见 include/linux/ide.h：

==================== include/linux/ide.h 620 620 ====================

```
#define DRIVER(drive)	((ide_driver_t *)((drive)->driver))
```

如前所述，ide_drive_t 数据结构中有个指针 driver，指向一个 ide_driver_t 数据结构（注意不要混淆这两种不同的数据结构），这就是具体设备的驱动函数跳转表。例如，一般 IDE 硬盘的函数表为 idedisk_driver，光盘驱动器的函数表为 ide_cdrom_driver，软盘的函数表为 idefloppy_driver，IDE 磁带驱动器的函数表为 idetape_driver。显然，这种数据结起起着类似于 file_operations 数据结构的作用，其类型定义见 include/linux/ide.h：

==================== include/linux/ide.h 599 618 ====================

```
typedef struct ide_driver_s {
	const char		*name;
	const char		*version;
```

```
	byte			media;
	unsigned busy		: 1;
	unsigned supports_dma		: 1;
	unsigned supports_dsc_overlap	: 1;
	ide_cleanup_proc	*cleanup;
	ide_do_request_proc		*do_request;
	ide_end_request_proc		*end_request;
	ide_ioctl_proc			*ioctl;
	ide_open_proc		*open;
	ide_release_proc	*release;
	ide_check_media_change_proc *media_change;
	ide_revalidate_proc		*revalidate;
	ide_pre_reset_proc	*pre_reset;
	ide_capacity_proc	*capacity;
	ide_special_proc	*special;
	ide_proc_entry_t	*proc;
} ide_driver_t;
```

数据结构 idedisk_driver 的值定义于 drivers/ide/ide-disk.c 中，为阅读方便再次把它列出如下：

==================== drivers/ide/ide-disk.c 711 733 ====================

```
/*
 *	IDE subdriver functions, registered with ide.c
 */
static ide_driver_t idedisk_driver = {
	"ide-disk",		/* name */
	IDEDISK_VERSION,	/* version */
	ide_disk,	/* media */
	0,		/* busy */
	1,		/* supports_dma */
	0,		/* supports_dsc_overlap */
	NULL,			/* cleanup */
	do_rw_disk,		/* do_request */
	NULL,			/* end_request */
	NULL,			/* ioctl */
	idedisk_open,		/* open */
	idedisk_release,	/* release */
	idedisk_media_change,	/* media_change */
	idedisk_revalidate,	/* revalidate */
	idedisk_pre_reset,	/* pre_reset */
	idedisk_capacity,	/* capacity */
	idedisk_special,	/* special */
	idedisk_proc		/* proc */
};
```

可见，其函数指针 do_request 指向 do_rw_disk()，这个函数的代码也在 drivers/ide/ide-disk.c 中。这又是一个很重要的函数，由于代码比较长，我们分段阅读。

==================== drivers/ide/ide-disk.c 377 422 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()>start_request()>do_rw_disk()]

```
/*
 * do_rw_disk() issues READ and WRITE commands to a disk,
 * using LBA if supported, or CHS otherwise, to address sectors.
 * It also takes care of issuing special DRIVE_CMDs.
 */
static ide_startstop_t do_rw_disk (ide_drive_t *drive, struct request *rq, unsigned long block)
{
	if (IDE_CONTROL_REG)
		OUT_BYTE(drive->ctl,IDE_CONTROL_REG);
	OUT_BYTE(rq->nr_sectors,IDE_NSECTOR_REG);
#ifdef CONFIG_BLK_DEV_PDC4030
	if (drive->select.b.lba || IS_PDC4030_DRIVE) {
#else /* !CONFIG_BLK_DEV_PDC4030 */
	if (drive->select.b.lba) {
#endif /* CONFIG_BLK_DEV_PDC4030 */
#ifdef DEBUG
		printk("%s: %sing: LBAsect=%ld, sectors=%ld, buffer=0x%08lx\n",
			drive->name, (rq->cmd==READ)?"read":"writ",
			block, rq->nr_sectors, (unsigned long) rq->buffer);
#endif
		OUT_BYTE(block,IDE_SECTOR_REG);
		OUT_BYTE(block>>=8,IDE_LCYL_REG);
		OUT_BYTE(block>>=8,IDE_HCYL_REG);
		OUT_BYTE(((block>>8)&0x0f)|drive->select.all,IDE_SELECT_REG);
	} else {
		unsigned int sect,head,cyl,track;
		track = block / drive->sect;
		sect  = block % drive->sect + 1;
		OUT_BYTE(sect,IDE_SECTOR_REG);
		head  = track % drive->head;
		cyl   = track / drive->head;
		OUT_BYTE(cyl,IDE_LCYL_REG);
		OUT_BYTE(cyl>>8,IDE_HCYL_REG);
		OUT_BYTE(head|drive->select.all,IDE_SELECT_REG);
#ifdef DEBUG
		printk("%s: %sing: CHS=%d/%d/%d, sectors=%ld, buffer=0x%08lx\n",
			drive->name, (rq->cmd==READ)?"read":"writ", cyl,
			head, sect, rq->nr_sectors, (unsigned long) rq->buffer);
#endif
	}
#ifdef CONFIG_BLK_DEV_PDC4030
	if (IS_PDC4030_DRIVE) {
		extern ide_startstop_t do_pdc4030_io(ide_drive_t *, struct request *);
		return do_pdc4030_io (drive, rq);
	}
#endif /* CONFIG_BLK_DEV_PDC4030 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里的 IDE_CONTROL_REG 定义见 include/linux/ide.h：

==================== include/linux/ide.h 126 126 ====================

```
#define IDE_CONTROL_REG     (HWIF(drive)->io_ports[IDE_CONTROL_OFFSET])
```

其它一些寄存器 I/O 地址的定义也与此相似。与前面 ide_do_request()代码中 1336 行调用的 SELECT_INTERRUPT 作一对比，就可以看出这一次的 bit1 为 0，表示解除对 IDE 接口中断请求的屏蔽。然后把需要读 / 写的扇区数量写入磁盘的“扇区数量寄存器”(386 行)。至于起始扇区，那就要看使用的是 LBA 模式（见 390 行）还是 CHS 模式（见 401 行）了。如果使用的是 CHS 模式就要作一些换算（403～404，406～407 行）。我们继续往下看：

==================== drivers/ide/ide-disk.c 423 431 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()>start_request()>do_rw_disk()]

```
423      if (rq->cmd == READ) {
424  #ifdef CONFIG_BLK_DEV_IDEDMA
425          if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_read, drive)))
426              return ide_started;
427  #endif /* CONFIG_BLK_DEV_IDEDMA */
428          ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
429          OUT_BYTE(drive->mult_count ? WIN_MULTREAD : WIN_READ, IDE_COMMAND_REG);
430          return ide_started;
431      }
```

大家知道，对外部设备的 I/O 技术有两种。一种是由 CPU 驱动的“程序控制 I/O”，其特点是：当外设已经准备好进行 I/O 时，就由 CPU 执行一段底层 I/O 驱动程序，在外设与内存缓冲区之间“搬运”数据，而外设与内存并不直接接触。另一种是由外设“直接访问内存”，即 DMA，CPU 把缓冲区的地址与需要读 / 写的长度告诉外设，外设在准备好以后就通过有关硬件向 CPU 发出一个 DMA 请求，要求 CPU 暂停使用内存，获得同意之后就直接在内存与外设之间传输数据，完成以后再把对内存的访问权归还给 CPU。但是，CPU 对曾经发生过的暂停使用内存以及谁在暂停期间访问了内存这些事情并无知觉，因为那是由硬件实现而不是在程序控制下实现的。所以，需要有一种手段，让 CPU 知道由某种设备驱动的一次 DMA 已经完成。那么，在程序控制 I/O 中怎样让 CPU 知道外设已经准备好，在 DMA 中怎样让 CPU 知道 DMA 已经完成呢？这又有两种方法，一种是由 CPU 查询（poll），另一种是由设备向 CPU 发出中断请求。这样，一共就有 4 种可能的组合。由 CPU 查询的方法显然是效率很低的，所以只在很简单、要求很低的系统中才使用，像 Linux 这样的系统当然要采用中断方法（见第 3 章）。于是就只剩下中断与 DMA 和中断与程序控制 I/O 两种了。是否采用 DMA 是一个系统配置的选项。如果选择了采用 DMA，这里的条件编译控制 CONFIG_BLK_DEV_IDEDMA 就有定义，否则就无定义，而在编译时跳过这里的 425 和 426 两行。我们将在以后专门讨论 DMA，在这里假定采用程序控制 I/O 的方式。

对于读操作，CPU 需要在磁盘已经准备好供读出时得到通知，再执行从磁盘读出，所以要预先设置好一个中断服务程序 read_intr()，这是由 ide_set_handler()完成的，它的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 525 548 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()

>ide_do_request()>start_request>do_rw_disk()>ide_set_handler()]

```
/*
 * This should get invoked any time we exit the driver to
 * wait for an interrupt response from a drive.  handler() points
 * at the appropriate code to handle the next interrupt, and a
 * timer is started to prevent us from waiting forever in case
 * something goes wrong (see the ide_timer_expiry() handler later on).
 */
void ide_set_handler (ide_drive_t *drive, ide_handler_t *handler,
               unsigned int timeout, ide_expiry_t *expiry)
{
     unsigned long flags;
     ide_hwgroup_t *hwgroup = HWGROUP(drive);

     spin_lock_irqsave(&io_request_lock, flags);
     if (hwgroup->handler != NULL) {
          printk("%s: ide_set_handler: handler not null; old=%p, new=%p\n",
               drive->name, hwgroup->handler, handler);
     }
     hwgroup->handler   = handler;
     hwgroup->expiry         = expiry;
     hwgroup->timer.expires = jiffies + timeout;
     add_timer(&hwgroup->timer);
     spin_unlock_irqrestore(&io_request_lock, flags);
}
```

这里的第一和第二个参数是不言自明的；第三个参数是为本次操作所设置的时间限制；第四个参数是个函数指针，这是当超过时间限制时要调用的函数，指针为 NULL 表示采用标准的超时处理。这些参数都记录在代表着具体硬盘所属磁盘组的 ide_hwgroup_t 数据结构中。从这个函数也可以看出，一个磁盘组在同一时间里只能有一个磁盘处于操作状态，因为在 ide_hwgroup_t 数据结构中只有一个函数指针指向中断服务程序，也只有一个用于操作超时的定时器。

为中断作好准备以后，CPU 这一边就万事俱备，可以通过命令寄存器下达启动读操作的命令了（见 429 行）。由于具体磁盘的内部缓冲区大小各不相同，有的硬盘每读出一个扇区就向 CPU 发出一个中断请求，有些则可以在积累起多个扇区的内容（如果需要的话）以后才向 CPU 发出一个中断请求。对后一种硬盘其 ide_drive_t 结构中的 mult_count 字段为非 0，而硬盘除接受读单个扇区的命令 WIN_READ 外也可接受读多个扇区的命令 WIN_MULTREAD。那么，所谓“多个扇区”，到底是几个呢？这就取决于需要读的扇区总数（见 386 行）和磁盘内部缓冲区的大小。这个缓冲区大小在 IDE 设备初始化时设置在 mult_count 字段中。读命令一经发出，对读操作的启动就完成了（430 行），所以返回一个常数 ide_started。从指定的扇区将数据读入其内部缓冲器后，磁盘就会向 CPU 发出中断请求，以后就是中断服务程序 read_intr()的事了。

再看写操作的启动：

==================== drivers/ide/ide-disk.c 432 476 ====================

[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()>start_request()>do_rw_disk()]

```
    if (rq->cmd == WRITE) {
        ide_startstop_t startstop;
#ifdef CONFIG_BLK_DEV_IDEDMA
        if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_write, drive)))
            return ide_started;
#endif /* CONFIG_BLK_DEV_IDEDMA */
        OUT_BYTE(drive->mult_count ? WIN_MULTWRITE : WIN_WRITE, IDE_COMMAND_REG);
        if (ide_wait_stat(&startstop, drive, DATA_READY, drive->bad_wstat, WAIT_DRQ)) {
            printk(KERN_ERR "%s: no DRQ after issuing %s\n", drive->name,
                drive->mult_count ? "MULTWRITE" : "WRITE");
            return startstop;
        }
        if (!drive->unmask)
            __cli(); /* local CPU only */
        if (drive->mult_count) {
            ide_hwgroup_t *hwgroup = HWGROUP(drive);
            /*
             * Ugh.. this part looks ugly because we MUST set up
             * the interrupt handler before outputting the first block
             * of data to be written.  If we hit an error (corrupted buffer list)
             * in ide_multwrite(), then we need to remove the handler/timer
             * before returning.  Fortunately, this NEVER happens (right?).
             *
             * Except when you get an error it seems...
             */
            hwgroup->wrq = *rq; /* scratchpad */
            ide_set_handler (drive, &multwrite_intr, WAIT_CMD, NULL);
            if (ide_multwrite(drive, drive->mult_count)) {
                unsigned long flags;
                spin_lock_irqsave(&io_request_lock, flags);
                hwgroup->handler = NULL;
                del_timer(&hwgroup->timer);
                spin_unlock_irqrestore(&io_request_lock, flags);
                return ide_stopped;
            }
        } else {
            ide_set_handler (drive, &write_intr, WAIT_CMD, NULL);
            idedisk_output_data(drive, rq->buffer, SECTOR_WORDS);
        }
        return ide_started;
    }
    printk(KERN_ERR "%s: bad command: %d\n", drive->name, rq->cmd);
    ide_end_request(0, HWGROUP(drive));
    return ide_stopped;
}
```

同样，我们在这里先不关心 DMA。对写操作也要先发出启动命令。同样，根据具体磁盘的特性，有针对单个扇区的 WIN_WRITE 和多个扇区的 WIN_MULTWRITE 两种命令。发出命令以后，就要通

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

过 ide_wait_stat()等待和读取磁盘的状态寄存器。这个等待只不过是大约 400 微毫秒的时间，所以不值得采用中断方式（磁盘也并不为此提供中断请求的功能）。另一方面，由于对 I/O 的启动是在 bh 函数中进行的，所以也不能进入睡眠。磁盘在接收到写操作命令并准备好接受待写入的数据时，就将其状态寄存器中的 DRQ 位设成 1，表示可以传输数据了。CPU 一方检测到这个信息以后，就可以开始输出数据了。可想而知，单扇区的写出与多扇区的写出会略有不同，我们把多扇区的写出（447～466 行）留给读者。这里只看与单扇区有关的代码。

首先还是设置中断服务程序，不过这一次设置的是 write_intr()。然后就是数据的输出了，函数 idedisk_output_data()的代码也在 drivers/ide/ide-disk.c 中：

==================== drivers/ide/ide-disk.c 79 87 ====================
[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()>ide_do_request()>start_request()>do_rw_disk()>idedisk_output_data()]

```
static inline void idedisk_output_data (ide_drive_t *drive, void *buffer, unsigned int wcount)
{
	if (drive->bswap) {
		idedisk_bswap_data(buffer, wcount);
		ide_output_data(drive, buffer, wcount);
		idedisk_bswap_data(buffer, wcount);
	} else
		ide_output_data(drive, buffer, wcount);
}
```

先看参数，第一个参数指向代表具体硬盘的 ide_drive_t 结构；第二个参数指向缓冲区，操作请求结构中的指针 buffer 总是指向其缓冲区队列中第一个缓冲区的开头；第三个参数是待写数据的长度，这里固定为 SECTOR_WORDS，即一个扇区的长度（以 16 位字计算）。以前讲过，同一操作请求所涉及的扇区一定是连续的，但是他们的缓冲区却不一定连续，不过缓冲区的长度一定是扇区大小的整数倍。例如，将两个操作请求合并成一个时，它们的扇区一定是连续的，但是缓冲区却不连续。

这个函数所做的还不是真正的数据输出，那是由 ide_output_data()完成的。这里所做的是可能需要的对字节次序的转换。CPU 有 big_ending 和 little_ending 之分，16 位和 32 位的数据在这两种制式中字节次序不同。可是，磁盘上的数据制式不应该跟着 CPU 跑，而应该固定使用一种，所以有可能需要在读／写磁盘时加以转换。

函数 ide_output_data()的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 405 436 ====================
[run_task_queue()>__run_task_queue()>generic_unplug_device()>__generic_unplug_device()>do_ide_request()**>ide_do_request()>start_request()>do_rw_disk()>idedisk_output_data()>ide_output_data()]**

```
/*
 * This is used for most PIO data transfers *to* the IDE interface
 */
void ide_output_data (ide_drive_t *drive, void *buffer, unsigned int wcount)
{
	byte io_32bit = drive->io_32bit;

	if (io_32bit) {
#if SUPPORT_VLB_SYNC
```

```
414          if (io_32bit & 2) {
415               unsigned long flags;
416               __save_flags(flags);    /* local CPU only */
417               __cli();      /* local CPU only */
418               do_vlb_sync(IDE_NSECTOR_REG);
419               outsl(IDE_DATA_REG, buffer, wcount);
420               __restore_flags(flags);/* local CPU only */
421          } else
422  #endif /* SUPPORT_VLB_SYNC */
423               outsl(IDE_DATA_REG, buffer, wcount);
424      } else {
425  #if SUPPORT_SLOW_DATA_PORTS
426          if (drive->slow) {
427               unsigned short *ptr = (unsigned short *) buffer;
428               while (wcount--) {
429                    outw_p(*ptr++, IDE_DATA_REG);
430                    outw_p(*ptr++, IDE_DATA_REG);
431               }
432          } else
433  #endif /* SUPPORT_SLOW_DATA_PORTS */
434               outsw(IDE_DATA_REG, buffer, wcount<<1);
435      }
436  }
```

大部分 IDE 磁盘的数据寄存器都是 16 位的，有些老式的接口还不允许操作太快而不能使用“成串输出”指令 OUTS，但也有些新式 IDE 磁盘提供 32 位数据寄存器。对大多数目前在使用中的 IDE 磁盘而言，具体的输出是通过成串输出指令（见 434 行）完成的。输出的数据写入到磁盘的内部缓冲区中。

输出完成以后，对写操作的启动就完成了。磁盘在完成本次写入操作（将数据写入指定的扇区中）后会向 CPU 发出中断请求，以后就是中断服务程序 write_intr()的事情了。同样，这里也返回 ide_started（见上面的 471 行）。

回前面 start_request()的代码中（见 1189 行）。对于有特殊要求的磁盘，其 special_all 字段可能为非 0，所以会走另一条路线执行 do_special()。不过，这些特殊的处理都带有初始化的性质，因而执行了一遍以后就把相应的标志位清成 0，以后 special_all 就为 0 而走正常的路线了。函数 do_special()的代码也在 drivers/ide/ide.c 中，我们把它留给有兴趣的读者自己阅读。

至此，对块设备 I/O 的启动已经完成，以后的事就完全是异步的了。下面我们以读操作为例来考察其全过程，读者在理解了该操作的全过程以后自然不难把它推广到写操作。

磁盘从指定的扇区中将数据读入其内部缓冲区以后，就向 CPU 发出中断请求。在 IDE 设备初始化时，从 hwif_init()调用的 init_irq()中，已经通过 request_irq()为每个 IDE 磁盘组登记了总的中断服务程序 ide_intr()，所以 CPU 在响应中断时就经由内核的中断服务机制（见第 3 章）进入了 ide_intr()，它的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 1524 1626 ====================

[do_IRQ()>handle_IRQ_event()>ide_intr()]

```
1524 /*
1525  * entry point for all interrupts, caller does __cli() for us
```

```
 */
void ide_intr (int irq, void *dev_id, struct pt_regs *regs)
{
     unsigned long flags;
     ide_hwgroup_t *hwgroup = (ide_hwgroup_t *)dev_id;
     ide_hwif_t *hwif;
     ide_drive_t *drive;
     ide_handler_t *handler;
     ide_startstop_t startstop;

     spin_lock_irqsave(&io_request_lock, flags);
     hwif = hwgroup->hwif;

     if (!ide_ack_intr(hwif)) {
          spin_unlock_irqrestore(&io_request_lock, flags);
          return;
     }

     if ((handler = hwgroup->handler) == NULL || hwgroup->poll_timeout != 0) {
          /*
           * Not expecting an interrupt from this drive.
           * That means this could be:
           *   (1) an interrupt from another PCI device
           *   sharing the same PCI INT# as us.
           * or     (2) a drive just entered sleep or standby mode,
           *   and is interrupting to let us know.
           * or     (3) a spurious interrupt of unknown origin.
           *
           * For PCI, we cannot tell the difference,
           * so in that case we just ignore it and hope it goes away.
           */
#ifdef CONFIG_BLK_DEV_IDEPCI
          if (IDE_PCI_DEVID_EQ(hwif->pci_devid, IDE_PCI_DEVID_NULL))
#endif    /* CONFIG_BLK_DEV_IDEPCI */
          {
               /*
                * Probably not a shared PCI interrupt,
                * so we can safely try to do something about it:
                */
               unexpected_intr(irq, hwgroup);
#ifdef CONFIG_BLK_DEV_IDEPCI
          } else {
               /*
                * Whack the status register, just in case we have a leftover pending IRQ.
                */
               (void) IN_BYTE(hwif->io_ports[IDE_STATUS_OFFSET]);
#endif /* CONFIG_BLK_DEV_IDEPCI */
          }
          spin_unlock_irqrestore(&io_request_lock, flags);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                return;
        }
        drive = hwgroup->drive;
        if (!drive) {
                /*
                 * This should NEVER happen, and there isn't much we could do about it here.
                 */
                spin_unlock_irqrestore(&io_request_lock, flags);
                return;
        }
        if (!drive_is_ready(drive)) {
                /*
                 * This happens regularly when we share a PCI IRQ with another device.
                 * Unfortunately, it can also happen with some buggy drives that trigger
                 * the IRQ before their status register is up to date.  Hopefully we have
                 * enough advance overhead that the latter isn't a problem.
                 */
                spin_unlock_irqrestore(&io_request_lock, flags);
                return;
        }
        if (!hwgroup->busy) {
                hwgroup->busy = 1; /* paranoia */
                printk("%s: ide_intr: hwgroup->busy was 0 ??\n", drive->name);
        }
        hwgroup->handler = NULL;
        del_timer(&hwgroup->timer);
        spin_unlock(&io_request_lock);

        if (drive->unmask)
                ide__sti();    /* local CPU only */
        startstop = handler(drive);      /* service this interrupt, may set handler for next interrupt */
        spin_lock_irq(&io_request_lock);

        /*
         * Note that handler() may have set things up for another
         * interrupt to occur soon, but it cannot happen until
         * we exit from this routine, because it will be the
         * same irq as is currently being serviced here, and Linux
         * won't allow another of the same (on any CPU) until we return.
         */
        set_recovery_timer(HWIF(drive));
        drive->service_time = jiffies - drive->service_start;
        if (startstop == ide_stopped) {
                if (hwgroup->handler == NULL) {  /* paranoia */
                        hwgroup->busy = 0;
                        ide_do_request(hwgroup, hwif->irq);
                } else {
                        printk("%s: ide_intr: huh? expected NULL handler on exit\n", drive->name);
                }
```

```
1624      }
1625      spin_unlock_irqrestore(&io_request_lock, flags);
1626 }
```

向内核的中断机制登记中断服务程序时，可以同时为之设置一个 void 指针，作为将来调用该服务程序时的参数（见第 3 章）。这就是这里的指针 dev_id，它实际上是指向磁盘组的 ide_hwgroup_t 数据结构，从这个数据结构中可以找到该磁盘组的当前 IDE 接口（见 1537 行）。对于 i386 结构的 CPU，这里的 ide_ack_intr()是个空函数，见 include/asm-i386/ide.h：

==================== include/asm-i386/ide.h 109 109 ====================

```
#define ide_ack_intr(hwif)       (1)
```

由于我们已经在前面的 do_rw_disk()中设置好了具体的中断服务程序，对于读操作是 read_intr()，这里的函数指针 hwgroup->handler 不可能为 0，所以我们跳过第 1544 行条件语句的执行部分。同时，1578 行的指针 hwgroup->drive 也不会是 0，因为磁盘之所以会发出中断请求是因为曾经向它发出了命令。但是，即使对于在正常情况下不会发生的事也要作好准备，以防万一。同样的道理也适用于 1585 行对磁盘状态的测试。

函数指针 hwgroup->handler 的内容已经转移到另一个指针 handler 中，1599 行将其清 0。这说明在磁盘组层次上中断服务程序的设置是一次性的，以后的中断服务程序是什么要视本次中断服务中的处理而定。向磁盘发出命令启动其写操作时，曾经为磁盘的操作设置下一个定时器，以备在操作超过时间限制时采取必要的措施。现在中断既已发生（并且已经检查了状态寄存器），这个定时器显然已经不需要了，所以通过 del_timer()将其撤销（1600 行）。

接着就是调用预先设置好的中断服务程序了（1605 行），对于读盘操作这是 read_intr()，其代码在 drivers/ide/ide-disk.c 中：

==================== drivers/ide/ide-disk.c 134 182 ====================

[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()]

```
/*
 * read_intr() is the handler for disk read/multread interrupts
 */
static ide_startstop_t read_intr (ide_drive_t *drive)
{
     byte stat;
     int i;
     unsigned int msect, nsect;
     struct request *rq;

     /* new way for dealing with premature shared PCI interrupts */
     if (!OK_STAT(stat=GET_STAT(),DATA_READY,BAD_R_STAT)) {
          if (stat & (ERR_STAT|DRQ_STAT)) {
               return ide_error(drive, "read_intr", stat);
          }
          /* no data yet, so wait for another interrupt */
          ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
          return ide_started;
     }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
153        msect = drive->mult_count;
154
155  read_next:
156        rq = HWGROUP(drive)->rq;
157        if (msect) {
158            if ((nsect = rq->current_nr_sectors) > msect)
159                nsect = msect;
160            msect -= nsect;
161        } else
162            nsect = 1;
163        idedisk_input_data(drive, rq->buffer, nsect * SECTOR_WORDS);
164  #ifdef DEBUG
165        printk("%s:  read: sectors(%ld-%ld), buffer=0x%08lx, remaining=%ld\n",
166            drive->name, rq->sector, rq->sector+nsect-1,
167            (unsigned long) rq->buffer+(nsect<<9), rq->nr_sectors-nsect);
168  #endif
169        rq->sector += nsect;
170        rq->buffer += nsect<<9;
171        rq->errors = 0;
172        i = (rq->nr_sectors -= nsect);
173        if (((long)(rq->current_nr_sectors -= nsect)) <= 0)
174            ide_end_request(1, HWGROUP(drive));
175        if (i > 0) {
176            if (msect)
177                goto read_next;
178            ide_set_handler (drive, &read_intr, WAIT_CMD, NULL);
179                return ide_started;
180        }
181          return ide_stopped;
182  }
```

这里的 OK_STAT()是个宏操作，它将由 GET_STAT()从状态寄存器读回的数值与 DATA_READY 和 BAD_R_STAT 比较，看是否状态字节中的 DATA_READY 位为 1 而 BAD_R_STAT 位为 0。如果状态字节表明磁盘尚未准备好，但是也没有出错（见 150 行），就再把中断服务程序设置成 read_intr()，到下次中断时再来处理。

如果磁盘确已准备好了，那就从相应的 ide_hwgroup_t 结构中找到当前的操作请求。以前讲过：每个 IDE 设备都有自己的操作请求队列，但是整个 IDE 接口组只能有一个“当前操作请求”，ide_hwgroup_t 结构中的指针 rq 就指向这个请求的 request 数据结构。所以，这里先要通过 HWGROUP()找到设备所属的接口组，再取得其当前操作请求。此外，还要根据反映磁盘内部缓冲区大小的 drive->mult_count 和本次操作中要求从磁盘读入当前缓冲区的扇区个数 rq->current_nr_sectors，确定一次从磁盘读入的扇区个数为 nsect。

函数 idedisk_input_data() 的代码在 drivers/ide/ide-disk.c 中。可以想像，它与前面的 idedisk_output_data()相似而只是传输数据的方向相反，所以我们不看具体的代码了。

从磁盘读入以后，要对当前操作请求的指针 buffer 和计数器 sector、nr_sectors、以及 current_nr_sectors 作相应的调整（见 169～170 以及 172～173 行），并计算出当前操作尚未完成的扇区

PDF-XCHANGE Click to buy NOW! www.docu-track.com

数（172 行）。这里 sector 为已经读入的累计扇区数（对本次操作请求）；nr_sectors 为本次操作请求要读入的扇区总数：current_nr_sectors 为本次中断服务中要从磁盘读入到当前内存缓冲区的扇区数；而 nsect 则为本次操作中要求磁盘读入，但尚未读入到内存缓冲区的扇区数。即使磁盘支持多扇区操作，一个操作请求也可能要通过好几次中断服务才能完成。此外，一个操作请求也可能涉及多个缓冲区，所以在 request 数据结构中有个缓冲区队列；对分布在不同缓冲区中的连续扇区可以合并成同一次多扇区操作，但是却要分次从磁盘读入，特别是考虑到采用 DMA 时更是如此，因为一般的 DMA 控制器（不是所谓 Intelegent DMA）没有自动切换缓冲区的功能。

考虑已经读进来的这部分数据与整体的关系，存在着三种可能的情况：

(1) 这些数据只是一部分，并且磁盘的内部缓冲区中还有数据尚未读出，之所以不能把内部缓冲区中的数据全部读出是因为已经到了一个内存缓冲段的末尾。在继续从磁盘的内部缓冲区读出之前，先要调整缓冲区指针使其指向下一个内存缓区，这是由 ide_end_request()完成的（见 173～174 行和 177 行）。

(2) 由于磁盘内部缓冲区大小的限制，磁盘内部缓冲区中的数据已经全部读出，但是当前操作尚未完成，磁盘会继续从有关扇区将数据读入内部缓冲区后再次发出中断请求，所以要为下一次中断作好准备（见 178～179 行）。同时，如果恰巧已经到了当前缓冲区的末尾，就也要通过 ide_end_request()启用新的缓冲区（见 173～174 行）。

(3) 这些数据就是当前操作请求所要求的全部，或者是它的最后一部分，所以整个操作请求已经完成（见 181 行）。

显然，当整个操作请求完成时，内存中的缓冲区指针也必定恰好到达一个缓冲区的末尾，所以也会调用 ide_end_request()（见 173 和 174 行）。

可见，只要到达了一个缓冲区的末尾，就会调用 ide_end_request()，其目的可能只是启用同一操作请求中的下一个缓冲区，也可能对整个操作请求的善后处理。它的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 505 523 ====================

[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()>ide_end_request()]

```
/*
 * This is our end_request replacement function.
 */
void ide_end_request (byte uptodate, ide_hwgroup_t *hwgroup)
{
     struct request *rq;
     unsigned long flags;

     spin_lock_irqsave(&io_request_lock, flags);
     rq = hwgroup->rq;

     if (!end_that_request_first(rq, uptodate, hwgroup->drive->name)) {
          add_blkdev_randomness(MAJOR(rq->rq_dev));
          blkdev_dequeue_request(rq);
          hwgroup->rq = NULL;
          end_that_request_last(rq);
     }
     spin_unlock_irqrestore(&io_request_lock, flags);
```

```
}
```

首先调用 end_that_request_first()，其代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 1099 1137 ====================
[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()>ide_end_request()>end_that_request_first()]

```
1099 /*
1100  * First step of what used to be end_request
1101  *
1102  * 0 means continue with end_that_request_last,
1103  * 1 means we are done
1104  */
1105
1106 int end_that_request_first (struct request *req, int uptodate, char *name)
1107 {
1108      struct buffer_head * bh;
1109      int nsect;
1110
1111      req->errors = 0;
1112      if (!uptodate)
1113           printk("end_request: I/O error, dev %s (%s), sector %lu\n",
1114                kdevname(req->rq_dev), name, req->sector);
1115
1116      if ((bh = req->bh) != NULL) {
1117           nsect = bh->b_size >> 9;
1118           req->bh = bh->b_reqnext;
1119           bh->b_reqnext = NULL;
1120           bh->b_end_io(bh, uptodate);
1121           if ((bh = req->bh) != NULL) {
1122                req->hard_sector += nsect;
1123                req->hard_nr_sectors -= nsect;
1124                req->sector = req->hard_sector;
1125                req->nr_sectors = req->hard_nr_sectors;
1126
1127                req->current_nr_sectors = bh->b_size >> 9;
1128                if (req->nr_sectors < req->current_nr_sectors) {
1129                     req->nr_sectors = req->current_nr_sectors;
1130                     printk("end_request: buffer-list destroyed\n");
1131                }
1132                req->buffer = bh->b_data;
1133                return 1;
1134           }
1135      }
1136      return 0;
1137 }
```

不管整个操作请求是否已经完成，对其中一个缓冲区的读入总归是完成了，否则就到不了这个函数中，所以把当前操作请求的缓冲区指针移向队列中的下一个缓冲区（1118 行），将原来的当前缓冲区从队列中解脱出来。同时，既然对一个缓冲区的操作已经完成，就要通过其函数指针 b_end_io 调用该

缓冲区的善后程序。对于写操作，这个函数指针是在__block_prepare_write()以及类似的函数中设置的；对于读操作则在 getblk()一类的函数中通过 init_buffer()设置，但是这个函数指针一般都指向 end_buffer_io_sync()，它的代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 978 985 ====================
[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()>ide_end_request()>end_that_request_first()>end_buffer_io_sync()]

```
/*
 * Default IO end handler, used by "ll_rw_block()".
 */
static void end_buffer_io_sync(struct buffer_head *bh, int uptodate)
{
	mark_buffer_uptodate(bh, uptodate);
	unlock_buffer(bh);
}
```

第一件事是将已经读／写完毕的缓冲区的 BH_Uptodate 标志位设成 1，表示该缓冲区的内容已是最新版本了。第二件事是 unlock_buffer()，见 focks.h：

==================== include/linux/locks.h 29 35 ====================
[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()>ide_end_request()>end_that_request_first()>end_buffer_io_sync()>unlock_buffer()]

```
extern inline void unlock_buffer(struct buffer_head *bh)
{
	clear_bit(BH_Lock, &bh->b_state);
	smp_mb__after_clear_bit();
	if (waitqueue_active(&bh->b_wait))
		wake_up(&bh->b_wait);
}
```

它一方面清除缓冲区的 BH_lock 标志位，一方面唤醒被锁在外面、正在睡眠等待的进程。哪些进程会被唤醒呢？至少启动了对该缓冲区的读／写操作，并调用了 wait_on_buffer()正在等待其完成的进程会被唤醒。例如，在 block_write()和 block_read()的代码中，当前进程在通过 ll_rw_block()发出对块设备的读／写请求以后都要调用 wait_on_buffer()等待。所以，wait_on_buffer()是启动操作的进程与异步的设备操作过程的同步点。当读／写操作涉及多个缓冲区时，对这个函数的调用通常放在一个循环中，以取得与所有缓冲区（读／写）的同步。

回到 end_that_request_first()的代码中。如果缓冲区队列中还有下一个缓冲区存在，就相应地设置 request 数据结构中的各项参数，为下一个缓冲区的读／写作好准备，并返回 1。如果缓冲区队列已经空了就返回 0，说明对整个操作请求的服务都已完成。

回到 ide_end_request()的代码中。如果 end_that_request_first()返回的是 1，就表示同一操作请求的缓冲区队列中还有下一个缓冲区，并且已经启用。既然操作尚未完成，刚到 read_intr()中自会进一步加以处理。反之，如果返回的是 0，那就说明整个操作请求都已完成了。这时候需要执行一些对整个操作请求的善后处理。处理些什么呢？一是通过 blkdev_dequeue_request()将代表这个操作请求的 request 结构从操作请求队列中摘除，并将整个 IDE 接口组的当前请求指针设置成 NULL。二是调用

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

end_that_request_last()，等一下我们要看它的源代码。除此之外，还搭配了一点“私货”，就是对 add_blkdev_randomness()的调用。这是干什么用的呢？系统中常常需要生成一些随机数，为了使这些“随机数”更加随机，需要在生成的过程中介入尽可能多、尽可能随机的因素，称为“熵”。例如，键盘输入的内容就可以用来作为这些因素之一，因为每次开机以后键盘输入的内容和次序往往是不完全一样的。而对块设备的访问也可以用来作为一个因素，这就是这里调用这个函数的原因。

函数 end_that_request_last()的代码在 drivers/block/ll_rw_blk.c 中：

==================== drivers/block/ll_rw_blk.c 1139 1149 ====================

[do_IRQ()>handle_IRQ_event()>ide_intr()>read_intr()>ide_end_request()>end_that_request_last()]

```
1139 void end_that_request_last(struct request *req)
1140 {
1141     if (req->e) {
1142         printk("end_that_request_last called with non-dequeued req\n");
1143         BUG();
1144     }
1145     if (req->sem != NULL)
1146         up(req->sem);
1147
1148     blkdev_release_request(req);
1149 }
```

显然，这里主要的操作是释放 request 结构（1148 行）和对内核信号量 req->sem 的 up()操作。用户进程可能通过系统调用 ioctl()直接启动一些对 IDE 设备的操作，对这些操作要通过使用内核信号量来保证互斥，现在既然操作已经完成就要通过 up()退出临界区。

回到 read_intr()中（175 行），如果还有扇区尚未读入，那就是因为原来的缓冲区已经满了，现在已经通过 ide_end_request()在 end_that_request_first()中切换到了下一个缓冲区，所以可以继续了。此时若 msect 非 0，就表示磁盘的内部缓冲区中还有数据尚未读出，所以直接转到标号 read_next 处（155 行）继续读出。否则，就要调用 ide_set_handler()再次设置中断服务程序，为下一次中断作好准备，这个函数的代码已经在前面看到过了。

当 CPU 从 read_intr()返回到 ide_intr()中时（见 ide_intr()中的 1606 行），其返回值有两种可能。一种是 ide_started，表示对当前操作请求的服务尚未完成，所以已经为下一次中断请求设置好了中断服务请求。不过，只要还没有从 ide_intr()返回，就不会因新的中断请求而嵌套进入这个服务程序。另一种是 ide_stopped，表示当前操作已经完成，代表当前操作请求的 request 数据结构也已经从操作请求队列中摘除。不管返回的值是什么，有两件事总是要做的。第一件事是通过 set_recovery_timer()根据当前的时间设置相应 ide_hwif_t 数据结构中的 last_time 字段。不过这只有对那些在两次操作之间需要有一个恢复时间的磁盘才必须。第二件事是根据当前时间和本次操作的启动时间计算出本次操作已经耗费的时间 service_time。

如果从 read_intr()返回的值是 ide_stopped，那就要调用 ide_do_request()，看看同一 IDE 接口组中是否还有哪个磁盘的队列中还有操作请求。这个函数的代码已经在前面读过了。

就这样，一直要到整个 IDE 接口组中再没有任何活跃（不在睡眠中）的磁盘还有操作请求时为止，到那时由中断驱动的设备读／写过程才会结束。如果结束时有磁盘正在睡眠中，而该磁盘的操作请求队列中又有操作请求，那么在睡眠到点时又会启动这个过程。当然，新操作请求的到来也会再次启动

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这个过程。

在前面的叙述和讨论中，我们假定采用的是程序控制 I/O 而不是 DMA，现在我们再来看看对 IDE 硬盘的 DMA 输入 / 输出是怎样实现的。如前所述，是否采用 DMA 是一个系统配置的选项，由条件编译控制量 CONFIG_BLK_DEV_IDEDMA 决定实际采用的代码。

大家知道，DMA 是“直接访问内存”的缩写，表示由外部设备直接访问内存。但是，在传统的 PC 系统结构中，DMA 操作要通过一个“ DMA 控制器”（更确切地说是 DMA 控制器中的一个“通道”）才能进行。在这种系统结构中，要对外设进行 DMA 操作时，DMA 控制器（经过 CPU 的设置和启动）就暂时“接管 ” CPU 的角色，成为内存的“主设备”，代替 CPU 发出访问内存和外设寄存器所需的地址以及控制脉冲，只是操作的速度比 CPU 更快。严格地说，在这种系统结构中，外设并没有变成内存的“主设备”而“当家作主”，从而“直接”访问内存，而是仍然处于被动的地位，只不过“主设备 ”从 CPU 变成了 DMA 控制器而已。所以，从外部设备的角度而言，比之别的一些系统结构，传统 PC 系统结构中的 DMA 有些名不副实。然而，PCI 总线的设计推广了 PC 系统结构中 DMA 操作的概念。以前我们曾提到，PCI 总线允许连接的总线上的设备竞争成为总线主设备，并且可以把内存看成是总线上的从设备。这样一来，PCI 设备就有可能真正当家作主，作为主设备来直接访问内存了。当然，有这种能力的 PCI 设备接口的结构要更复杂一些。因具体设备性质的不同，PCI 设备（接口）可以做成具有成为总线主设备的能力，也可以不具备这种能力。对于可以成为总线主设备的设备，在配置寄存器组的命令寄存器中有个控制位，可以允许或不允许该设备（在需要时）参加竞争。IDE 硬盘（接口）就具有成为总线主设备的能力，因而可以进行真正意义上的 DMA 操作。为了与传统 PC 系统结构的 DMA 操作相区别，这种 DMA 操作称为“总线主 DMA”（Bus Master DMA）或 BMDMA。至于不具备 BMDMA 功能的 PCI 设备，如果有必要的话，仍可以通过 DMA 控制器对其进行“ DMA”操作。

为了 BMDMA 的需要，IDE 接口中增设了两组 DMA 寄存器，分别用于 IDE 接口上的两个通道。这些寄存器采用 I/O 地址，并且固定为 IDE 接口的第五个地址区间（配置寄存器组中的 resource[4]）。而且，IDE 接口不但能进行传统的、单缓冲区的 DMA，还能进行多缓冲区间（一个区间中可以包含若干连续的缓冲区）“串式 ”DMA。只要为之准备下一个“ DMA 区间表”，表中逐个地列出用于 DMA 操作的若干缓冲区间（包括起点与长度），并把这个表的起始地址写入 IDE 接口上相应的“总线主 IDE 描述表指针”寄存器（BMIDTPX），则启动 DMA 操作以后 IDE 接口会先从表中找到第一个缓冲区间的地址；完成对第一个缓冲区间的操作以后就会自动转到第二个缓冲区间；如此等等，直到完成对所有缓冲区间的操作。下面读者就会看到，一旦准备好了 DMA 区间表，实现 DMA 操作的代码就很简单了。

采用 DMA 与否只涉及很底层的代码，对具体设备驱动程序的结构并无显著的影响。对于 IDE 硬盘的驱动，区别主要在底层函数 do_rw_disk()中，为方便阅读，我们再把这个函数的有关片断列出于下（drivers/ide/ide-disk.c）：

==================== drivers/ide/ide-disk.c 377 383 ====================

```
/*
 * do_rw_disk() issues READ and WRITE commands to a disk,
 * using LBA if supported, or CHS otherwise, to address sectors.
 * It also takes care of issuing special DRIVE_CMDs.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
381  */
382 static ide_startstop_t do_rw_disk (ide_drive_t *drive, struct request *rq, unsigned long block)
383 {
     . . . . . .
```

==================== drivers/ide/ide-disk.c 423 437 ====================

```
423     if (rq->cmd == READ) {
424 #ifdef CONFIG_BLK_DEV_IDEDMA
425         if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_read, drive)))
426             return ide_started;
427 #endif /* CONFIG_BLK_DEV_IDEDMA */
428         ide_set_handler(drive, &read_intr, WAIT_CMD, NULL);
429         OUT_BYTE(drive->mult_count ? WIN_MULTREAD : WIN_READ, IDE_COMMAND_REG);
430         return ide_started;
431     }
432     if (rq->cmd == WRITE) {
433         ide_startstop_t startstop;
434 #ifdef CONFIG_BLK_DEV_IDEDMA
435         if (drive->using_dma && !(HWIF(drive)->dmaproc(ide_dma_write, drive)))
436             return ide_started;
437 #endif /* CONFIG_BLK_DEV_IDEDMA */
     . . . . . .
```

==================== drivers/ide/ide-disk.c 476 476 ====================

```
476 }
```

在 423 行以前的代码已经对 IDE 硬盘的有关寄存器（如要读 / 写的块号等）进行了必要的设置。有关详情可参看前面对程序控制 I/O 方式的说明。现在，剩下的只是对读 / 写操作的启动了。正是在这一步上，DMA 与程序控制 I/O 有了区别。

当采用 DMA 时，对 IDE 硬盘的读 / 写操作都是通过 ide_hwif_t 数据结构中提供的函数指针 dmaproc 完成，只是参数 func 一为 ide_dma_read，一为 idc_dma_write。这个函数指针在初始化时设置成指向 ide_dmaproc()，其代码在 drivers/ide/ide-dma.c 中：

==================== drivers/ide/ide-dma.c 447 543 ====================

[do_rw_disk()>ide_dmaproc()]

```
/*
 * ide_dmaproc() initiates/aborts DMA read/write operations on a drive.
 *
 * The caller is assumed to have selected the drive and programmed the drive's
 * sector address using CHS or LBA.  All that remains is to prepare for DMA
 * and then issue the actual read/write DMA/PIO command to the drive.
 *
 * For ATAPI devices, we just prepare for DMA and return. The caller should
 * then issue the packet command to the drive and call us again with
 * ide_dma_begin afterwards.
 *
 * Returns 0 if all went well.
 * Returns 1 if DMA read/write could not be started, in which case
 * the caller should revert to PIO for the current request.
 * May also be invoked from trm290.c
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 */
int ide_dmaproc (ide_dma_action_t func, ide_drive_t *drive)
{
    ide_hwif_t *hwif = HWIF(drive);
    unsigned long dma_base = hwif->dma_base;
    byte unit = (drive->select.b.unit & 0x01);
    unsigned int count, reading = 0;
    byte dma_stat;

    switch (func) {
        case ide_dma_off:
            printk("%s: DMA disabled\n", drive->name);
        case ide_dma_off_quietly:
            outb(inb(dma_base+2) & ~(1<<(5+unit)), dma_base+2);
        case ide_dma_on:
            drive->using_dma = (func == ide_dma_on);
            if (drive->using_dma)
                outb(inb(dma_base+2)|(1<<(5+unit)), dma_base+2);
            return 0;
        case ide_dma_check:
            return config_drive_for_dma (drive);
        case ide_dma_read:
            reading = 1 << 3;
        case ide_dma_write:
            SELECT_READ_WRITE(hwif,drive,func);
            if (!(count = ide_build_dmatable(drive, func)))
                return 1;/* try PIO instead of DMA */
            outl(hwif->dmatable_dma, dma_base + 4); /* PRD table */
            outb(reading, dma_base);             /* specify r/w */
            outb(inb(dma_base+2)|6, dma_base+2);      /* clear INTR & ERROR flags */
            drive->waiting_for_dma = 1;
            if (drive->media != ide_disk)
                return 0;
            ide_set_handler(drive, &ide_dma_intr, WAIT_CMD, dma_timer_expiry);    /* issue cmd to drive */
            OUT_BYTE(reading ? WIN_READDMA : WIN_WRITEDMA, IDE_COMMAND_REG);
        case ide_dma_begin:
            /* Note that this is done *after* the cmd has
             * been issued to the drive, as per the BM-IDE spec.
             * The Promise Ultra33 doesn't work correctly when
             * we do this part before issuing the drive cmd.
             */
            outb(inb(dma_base)|1, dma_base);     /* start DMA */
            return 0;
        case ide_dma_end: /* returns 1 on error, 0 otherwise */
            drive->waiting_for_dma = 0;
            outb(inb(dma_base)&~1, dma_base);    /* stop DMA */
            dma_stat = inb(dma_base+2);     /* get DMA status */
            outb(dma_stat|6, dma_base+2);   /* clear the INTR & ERROR bits */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
510                 ide_destroy_dmatable(drive);     /* purge DMA mappings */
511                 return (dma_stat & 7) != 4; /* verify good DMA status */
512            case ide_dma_test_irq: /* returns 1 if dma irq issued, 0 otherwise */
513                 dma_stat = inb(dma_base+2);
514  #if 0     /* do not set unless you know what you are doing */
515                 if (dma_stat & 4) {
516                      byte stat = GET_STAT();
517                      outb(dma_base+2, dma_stat & 0xE4);
518                 }
519  #endif
520                 return (dma_stat & 4) == 4; /* return 1 if INTR asserted */
521            case ide_dma_bad_drive:
522            case ide_dma_good_drive:
523                 return check_drive_lists(drive, (func == ide_dma_good_drive));
524            case ide_dma_verbose:
525                 return report_drive_dmaing(drive);
526            case ide_dma_timeout:
527  #ifdef CONFIG_BLK_DEV_IDEDMA_TIMEOUT
528                 /*
529                  * Have to issue an abort and requeue the request
530                  * DMA engine got turned off by a goofy ASIC, and
531                  * we have to clean up the mess, and here is as good
532                  * as any.  Do it globally for all chipsets.
533                  */
534  #endif /* CONFIG_BLK_DEV_IDEDMA_TIMEOUT */
535            case ide_dma_retune:
536            case ide_dma_lostirq:
537                 printk("ide_dmaproc: chipset supported %s func only: %d\n", ide_dmafunc_verbose(func),
func);
538                 return 1;
539            default:
540                 printk("ide_dmaproc: unsupported %s func: %d\n", ide_dmafunc_verbose(func), func);
541                 return 1;
542       }
543  }
```

我们在这里只关心 ide_dma_read 和 ide_dma_write 两种操作，注意在 484 行下面没有 break 或 return 语句，读和写的区别仅在于变量 reading 的值为 0x8 或 0x0，IDE 硬盘接口的 BMDMA 命令寄存器中用这一位区分操作的方向。此外，在 496 行后面也没有 break 或 return 语句，也就是说 ide_dma_read 或 ide_dma_write 操作都蕴含着 ide_dma_begin 操作。宏操作 SELECT_READ_WRITE 定义于 include/linux/ide.h：

==================== include/linux/ide.h 203 207 ====================

```
203  #define SELECT_READ_WRITE(hwif,drive,func)              \
204  {                                             \
205       if (hwif->rwproc)                             \
206            hwif->rwproc(drive,func);                    \
207  }
```

其目的是为一些特殊的 IDE 硬盘提供可能需要的附加操作，一般的 IDE 硬盘不需要附加操作，这个函数指针 rwproc 就为 0。

启动 DMA 操作之前，先要通过 ide_build_dmatable()准备好 DMA 区间表。这个函数的代码在 drivers/ide/ide-dma.c 中：

==================== drivers/ide/ide-dma.c 248 309 ====================

[do_rw_disk()>ide_dmaproc()>ide_build_dmatable()]

```
int ide_build_dmatable (ide_drive_t *drive, ide_dma_action_t func)
{
     unsigned int *table = HWIF(drive)->dmatable_cpu;
#ifdef CONFIG_BLK_DEV_TRM290
     unsigned int is_trm290_chipset = (HWIF(drive)->chipset == ide_trm290);
#else
     const int is_trm290_chipset = 0;
#endif
     unsigned int count = 0;
     int i;
     struct scatterlist *sg;

     HWIF(drive)->sg_nents = i = ide_build_sglist(HWIF(drive), HWGROUP(drive)->rq);

     sg = HWIF(drive)->sg_table;
     while (i && sg_dma_len(sg)) {
          u32 cur_addr;
          u32 cur_len;

          cur_addr = sg_dma_address(sg);
          cur_len = sg_dma_len(sg);

          /*
           * Fill in the dma table, without crossing any 64kB boundaries.
           * Most hardware requires 16-bit alignment of all blocks,
           * but the trm290 requires 32-bit alignment.
           */

          while (cur_len) {
               if (++count >= PRD_ENTRIES) {
                    printk("%s: DMA table too small\n", drive->name);
                    pci_unmap_sg(HWIF(drive)->pci_dev,
                              HWIF(drive)->sg_table,
                              HWIF(drive)->sg_nents,
                              HWIF(drive)->sg_dma_direction);
                    return 0; /* revert to PIO for this request */
               } else {
                    u32 xcount, bcount = 0x10000 - (cur_addr & 0xffff);

                    if (bcount > cur_len)
                         bcount = cur_len;
```

```
                    *table++ = cpu_to_le32(cur_addr);
                    xcount = bcount & 0xffff;
                    if (is_trm290_chipset)
                        xcount = ((xcount >> 2) - 1) << 16;
                    *table++ = cpu_to_le32(xcount);
                    cur_addr += bcount;
                    cur_len -= bcount;
                }
            }

            sg++;
            i--;
        }

        if (!count)
            printk("%s: empty DMA table?\n", drive->name);
        else if (!is_trm290_chipset)
            *--table |= cpu_to_le32(0x80000000);

        return count;
}
```

在 ide_hwif_t 数据结构中有两个指针，即 dmatable_cpu 和 dmatable_dma，二者都指向同一个用于 DMA 区间表的页面，这个页面是在初始化时分配好的。所不同的是，dmatable_cpu 通过页面的虚拟地址指向这个页面，这是 CPU 所看到的 DMA 区间表；而 dmatable_dma 则通过其物理地址指向这个页面，这是 IDE 接口的 DMA 功能部分所看到的 DMA 区间表。同时，ide_hwif_t 数据结构中还有个指针 sg_table，指向一个 scatterlist 结构数组。每个 scatterlist 结构都描述了一个用于 DMA 操作的缓冲区间，包括其起始（虚拟）地址 address 与长度 length。如前所述，每个缓冲区间可以包含若干连续的缓冲区，而 IDE 接口可以对若干个散布的（所以称为 scatterlist）缓冲区间操作。当文件系统层要求设备驱动层完成对具体设备的 I/O 时，交下来的是一个 buffer_head 结构队列，队列中的每个 buffer_head 数据结构都描述着一个缓冲区。这些缓冲区中有些是互相连续的，有些则不是，所以先要把它们整理成若干个缓冲区间，建立起一个 scatterlist，为进一步建立 DMA 区间表作好准备。这种数据结构定义于 include/asm-i386/scatterlist.h：

==================== include/asm-i386/scatterlist.h 4 9 ====================

```
struct scatterlist {
    char *  address;    /* Location data is to be transferred to */
    char * alt_address; /* Location of actual if address is a
                * dma indirect buffer.  NULL otherwise */
    unsigned int length;
};
```

先通过 ide_build_sglist()根据具体的操作请求建立起一个“物理区间表”，把同一个操作请求中互相连续的缓冲区都合并成缓冲区间，这个函数的代码在 drivers/ide/ide-dma.c 中：

==================== drivers/ide/ide-dma.c 214 241 ====================

```
static int ide_build_sglist (ide_hwif_t *hwif, struct request *rq)
{
	struct buffer_head *bh;
	struct scatterlist *sg = hwif->sg_table;
	int nents = 0;

	if (rq->cmd == READ)
		hwif->sg_dma_direction = PCI_DMA_FROMDEVICE;
	else
		hwif->sg_dma_direction = PCI_DMA_TODEVICE;
	bh = rq->bh;
	do {
		unsigned char *virt_addr = bh->b_data;
		unsigned int size = bh->b_size;

		while ((bh = bh->b_reqnext) != NULL) {
			if ((virt_addr + size) != (unsigned char *) bh->b_data)
				break;
			size += bh->b_size;
		}
		memset(&sg[nents], 0, sizeof(*sg));
		sg[nents].address = virt_addr;
		sg[nents].length = size;
		nents++;
	} while (bh != NULL);

	return pci_map_sg(hwif->pci_dev, sg, nents, hwif->sg_dma_direction);
}
```

这里的函数 pci_map_sg()实际上并没有什么操作，只是对参数的合理性作一些检验，然后返回 nents，即区间的个数。这样，操作请求所涉及的一串缓冲区就合并成了若干区间，由一个 scatterlist 结构数组代表。数组的大小为 nents，最后保存在 ide_hwif_t 结构的 sg_nents 字段中。不过，这个 scatterlist 结构数组还不是 DMA 区间表，所以在回到 ide_build_dmatable()中（260 行）还要据此建立起 DMA 区间表。

DMA 区间表也是以缓冲区间为基础的。但是，由于 IDE 接口中 DMA 控制部分的设计，缓冲区间不能跨越 64KB 的边界（显然，用来发出 32 位内存地址的寄存器分成两截，其高 16 位在整个 DMA 操作过程中固定不变）。因此，对于 scatterlist 结构数组中的每个区间，代码中通过一个 while 循环加以检查，如果跨越了 64KB 边界就要将其分割成若干 DMA 缓冲区间。与此同时，则根据最终形成的缓冲区间建立起 DMA 区间表。DMA 缓冲区间不能跨越 64KB 边界，如果缓冲区间的起点恰好与 64KB 边界对齐，则每个 DMA 缓冲区最大可达 64KB，否则就取决于起点的位置。例如，如果缓冲区的起点在 16KB 处，那么最大可达 48KB（见 285 行）。DMA 区间表是个 32 位无符号整数数组，数组中每两个无符号整数合在一起描述一个 DMA 缓冲区，称为“物理区间描述”（Physical Region Descriptor）。其中第一个无符号整数为缓冲区起点的物理地址（289 行），第二个为缓冲区的长度（293 行），均须转换成“ Little Ending”格式。区间的起点在一开始（267 行）就通过宏操作 sg_dma_address()转换成了物理地址，所以随后计算出来（294 行）的都是物理地址。宏操作 sg_dma_address()的定义在 include/asm-i386/pci.h

PDF-XCHANGE Click to buy NOW! www.docu-track.com

中：

==================== include/asm-i386/pci.h 164 165 ====================

```
#define sg_dma_address(sg)  (virt_to_bus((sg)->address))
#define sg_dma_len(sg)      ((sg)->length)
```

总线地址实际上就是物理地址，宏操作 virt_to_bus 也就是 virt_to_phys，定义于 include/asm-i386/io.h：

==================== include/asm-i386/io.h 157 161 ====================

```
/*
 * IO bus memory addresses are also 1:1 with the physical address
 */
#define virt_to_bus virt_to_phys
#define bus_to_virt phys_to_virt
```

为 DMA 区间表分配的空间是一个页面，分别用于 IDE 接口的两个设备，所以每个操作表的容量为 PRD_ENTRIES，即 256。如果要求一次就读／写超过 256 个 DMA 缓冲区（难以想像），那就不能通过 DMA 操作来完成了，此时 ide_build_dmatable()返回 0，从而使 ide_dmaproc()返回 1（见 488 行）。从 do_rw_disk()的代码中（425 和 435 行）可以看出，当 ide_dmaproc()返回 1 时，就又回到程序控制 I/O 方式的代码中继续执行，仍能完成所要求的读／写，只不过效率低一些而已。

回到 ide_dmaproc()的代码中（487 行），准备好 DMA 区间表以后，下面就是对 DMA 控制器中有关寄存器的操作了。为便于阅读，我们再列出这个函数中的关键几行：

==================== drivers/ide/ide-dma.c 489 504 ====================

[do_rw_disk()>ide_dmaproc()]

```
489              outl(hwif->dmatable_dma, dma_base + 4); /* PRD table */
490              outb(reading, dma_base);            /* specify r/w */
491              outb(inb(dma_base+2)|6, dma_base+2);      /* clear INTR & ERROR flags */
492              drive->waiting_for_dma = 1;
493              if (drive->media != ide_disk)
494                   return 0;
495              ide_set_handler(drive, &ide_dma_intr, WAIT_CMD, dma_timer_expiry);    /* issue cmd to drive */
496              OUT_BYTE(reading ? WIN_READDMA : WIN_WRITEDMA, IDE_COMMAND_REG);
497         case ide_dma_begin:
498              /* Note that this is done *after* the cmd has
499               * been issued to the drive, as per the BM-IDE spec.
500               * The Promise Ultra33 doesn't work correctly when
501               * we do this part before issuing the drive cmd.
502               */
503              outb(inb(dma_base)|1, dma_base);      /* start DMA */
504              return 0;
```

首先将 DMA 区间表的（物理）地址 hwif->dmatable_dma 写入描述表指针寄存器（489 行）。把操作的方向（读或写）写入其“命令寄存器”（490 行），并将状态寄存器中的“中断请求”和“出错”两个标志位清成 0（491 行）。IDE 接口的 DMA 控制部分共有 16 个字节，前 8 个用于接口上的第一个设

PDF-XCHANGE Click to buy NOW! www.docu-track.com

备 ide0，后 8 个则用于 ide1。根据具体的设备，初始化时就把相应 ide_hwif_t 数据结构中的字段 dma_base 设置成指向其 DMA 控制部分的起点，这就是这里的 dma_base（见 466 行）。其中从地址 dma_base 开始是 8 位的命令寄存器，从(dma_base+2)开始是 8 位的状态寄存器，从(dma_base+4)开始则是 32 位的描述表指针寄存器。

然后，通过 ide_set_handler()设置好本次 DMA 操作结束时的中断服务程序 ide_dma_intr()，同时也设置好超时处理程序 dma_timer_expiry()，ide_set_handler()的代码在 drivers/ide/ide.c 中：

==================== drivers/ide/ide.c 532 548 ====================

```
void ide_set_handler (ide_drive_t *drive, ide_handler_t *handler,
                unsigned int timeout, ide_expiry_t *expiry)
{
    unsigned long flags;
    ide_hwgroup_t *hwgroup = HWGROUP(drive);

    spin_lock_irqsave(&io_request_lock, flags);
    if (hwgroup->handler != NULL) {
        printk("%s: ide_set_handler: handler not null; old=%p, new=%p\n",
            drive->name, hwgroup->handler, handler);
    }
    hwgroup->handler   = handler;
    hwgroup->expiry        = expiry;
    hwgroup->timer.expires = jiffies + timeout;
    add_timer(&hwgroup->timer);
    spin_unlock_irqrestore(&io_request_lock, flags);
}
```

最后，向 IDE 硬盘的命令寄存器发出 WIN_READDMA 或 WIN_WRITEDMA，启动硬盘的操作（496 行），紧接着（503 行）向其 DMA 命令寄存器也发出启动命令（将其最低位设成 1）。此后，IDE 硬盘和 DMA 控制器的操作对于 CPU 就是透明的了。以读操作为例，IDE 硬盘在有了供读出的数据以后便向其 DMA 控制部分发出电信号。而 DMA 控制部分，则从 DMA 区间表中找到第一个缓冲区间。然后产生出反复从 IDE 接口读出数据并写入缓冲区间所需的电信号。如果一个缓冲区间满了，就从 DMA 区间表中找到下一个缓冲区间，再继续往里写。这样，一直要到完成了当前的整个 DMA 操作以后才会向 CPU 发出一个中断请求，使 CPU 转入 DMA 操作的中断服务程序 ide_dma_intr()中，这个函数的代码在 drivers/ide/ide-dma.c 中：

==================== drivers/ide/ide-dma.c 189 212 ====================

```
/*
 * dma_intr() is the handler for disk read/write DMA interrupts
 */
ide_startstop_t ide_dma_intr (ide_drive_t *drive)
{
    int i;
    byte stat, dma_stat;

    dma_stat = HWIF(drive)->dmaproc(ide_dma_end, drive);
    stat = GET_STAT();          /* get drive status */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (OK_STAT(stat,DRIVE_READY,drive->bad_wstat|DRQ_STAT)) {
        if (!dma_stat) {
            struct request *rq = HWGROUP(drive)->rq;
            rq = HWGROUP(drive)->rq;
            for (i = rq->nr_sectors; i > 0;) {
                i -= rq->current_nr_sectors;
                ide_end_request(1, HWGROUP(drive));
            }
            return ide_stopped;
        }
        printk("%s: dma_intr: bad DMA status\n", drive->name);
    }
    return ide_error(drive, "dma_intr", stat);
}
```

DMA 中断的发生意味着一次 DMA 操作的结束，因此以功能码 ide_dma_end 为参数调用 ide_dmaproc()。同样，为方便阅读，我们再把这个函数中有关的片断列出于下：

==================== drivers/ide/ide-dma.c 505 511 ====================
[ide_dma_intr()>ide_dmaproc()]

```
        case ide_dma_end: /* returns 1 on error, 0 otherwise */
            drive->waiting_for_dma = 0;
            outb(inb(dma_base)&~1, dma_base);     /* stop DMA */
            dma_stat = inb(dma_base+2);      /* get DMA status */
            outb(dma_stat|6, dma_base+2);    /* clear the INTR & ERROR bits */
            ide_destroy_dmatable(drive);     /* purge DMA mappings */
            return (dma_stat & 7) != 4; /* verify good DMA status */
```

首先将 DMA 控制器命令寄存器的最低位清 0，使其停止运行；然后读入其状态寄存器，并将其中断标志位 INTR 和出错标志位清 0。这里还调用了一个函数 ide_destroy_dmatable()，其代码在 drivers/ide/ide-dma.c 中：

==================== drivers/ide/ide-dma.c 311 319 ====================
[ide_dma_intr()>ide_dmaproc()>ide_destroy_dmatable()]

```
/* Teardown mappings after DMA has completed.  */
void ide_destroy_dmatable (ide_drive_t *drive)
{
    struct pci_dev *dev = HWIF(drive)->pci_dev;
    struct scatterlist *sg = HWIF(drive)->sg_table;
    int nents = HWIF(drive)->sg_nents;

    pci_unmap_sg(dev, sg, nents, HWIF(drive)->sg_dma_direction);
}
```

这里的 pci_unmap_sg()实际上并没有什么操作。其实，DMA 区间表也无须废除，用于 DMA 区间表的页面并不需要释放，下一次要进行 DMA 操作时自会再在同一个页面中建立起新的 DMA 区间表。

然后，如果状态寄存器指示操作并未出错，则对操作请求中涉及的缓冲区调用 ide_end_request()。这个函数的代码已经在前面讲述程序控制 I/O 时读过了，读者可以回过去复习一下。另一方面，这也

PDF-XCHANGE Click to buy NOW! www.docu-track.com

说明 DMA 操作和程序控制 I/O 殊途同归，都到了调用 ide_end_request()的时候，下面就都一样了。

上面我们以 IDE 硬盘为例说明了块设备的驱动。实际的块设备种类当然很多，光是 IDE 设备就有硬盘、软盘、磁带、光盘（CD_ROM）及可写光盘等。我们不可能在一本书中一一加以介绍。但是，只要理解了对 IDE 硬盘的驱动，读者在进一步阅读分析其它设备的驱动程序代码时就不至于有太大的问题了。此外，除 IDE 硬盘外，SCSI 硬盘也是很常用的，但是限于篇幅我们也不能对 SCSI 总线以及有关设备的驱动再作介绍了。有兴趣或需要的读者可以参考有关专著或技术资料，自行阅读有关的代码，这些代码大多在 drivers/scsi 目录下。

最后，还有一个非常值得一提的话题是磁盘阵列。在 drivers/md 目录下有 raid0.c、raid1.c 等文件是关于磁盘阵列的，但是限于篇幅也不能在本书中加以介绍，而只好留给读者了。读者也许对此感到惋惜，我们也感到遗憾。

8.6 字符设备驱动概述

在内核中，字符设备的驱动是最多样、最灵活多变的部分。这首先是因为字符设备本身的多样性，各种字符设备在作用、功能、结构等等方面真是五花八门。有的“字符设备”甚至并不是字面意义上的“设备”，如下面要讲到的/dev/null 就是一个例子；而有的则又相当复杂而需要把驱动程序进一步划分成若干子层，或者进一步分解成若干项低层的设备，如 PC 机的控制台终端实际上就包括了显示器和键盘。其次，这种多样性还来自对设计和开发字符设备驱动程序的广泛参与。全世界通过英特网参与 Linux 开发的人数以千计，如果说如进程、调度、进程间通信、内存管理等方面的工作相对而言还比较集中于某一些核心人士的话，那么对于设备驱动，特别是字符设备驱动方面的工作就分布得很广了。这种广泛的参与当然会带来一些风格上、技巧运用上、程序结构上的多样性。不过，由于 Linux 内核总体上的结构性和模块性，这种多样性只是次要的，并不占主导地位。最后，多样性还来自不同的历史渊源。其中最重要的是对一些（并非全部）网络设备（如 Ethernet 接口卡）的驱动，这些设备的驱动从一开始就纳入了“插口”（socket）机制的范畴而并不遵循 Unix/Linux 为设备驱动设计的统一的格局，那就是通过主设备号 / 次设备号来区分设备并且引导 CPU 进入具体设备的驱动程序。因此，这些设备似乎既不属于块设备，也不属于字符设备，其驱动方式自成一格，并且没有相应的“设备文件”。由于篇幅的限制，本书将不涉及网络方面的内容，而只专注于传统的设备驱动模式。尽管有些网络设备的驱动实际上确实遵循这种统一的格局，我们也只好从略。

正如我们反复讲过的那样，传统的、统一的 Unix 设备驱动是以主 / 次设备号为纲的。每项设备都属于块设备或字符设备，都有惟一的主设备号和次设备号，而内核中则有块设备表和字符设备表，根据设备的类型和主设备号便可通过这两个设备表之一找到相应的驱动函数跳转结构，而次设备号则一般只用作同类型设备中具体设备项的编号（通常决定着接口的 I/O 地址）。但是，由于字符设备的多样性，有时候也用次设备号作进一步的归类。这方面典型的例子就是终端设备 TTY。TTY 设备是字符设备，主设备号为 4，但是当次设备号为 0 时表示“当前虚拟控制终端”，而 1～63 表示 63 个可能的“虚拟控制终端”，64～255 则表示 192 个可能的串行口 UART（通用异步收发器）和连接在上面的实际终

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

端设备。这里所谓实际终端设备通常是指老式的 CRT 终端，或“笨终端”，而“虚拟控制终端”则通常是建立在 PC 机的显示器和图形接口卡基础上的。显然，在这里相同的主设备号并不意味着相同的驱动程序。由于控制终端的重要性和复杂性，我们将用两节的篇幅专门介绍它的驱动程序。另一个例子是主设备号为 1 的“字符设备”，当次设备号为 1 时表示物理内存/dev/mem，为 2 时则表示内核的虚存空间/dev/kmem，为 3 时就表示“空设备 ” /dev/null，而次设备号 8 则表示随机数生成器/dev/random。类似的情况在块设备中虽然也有（例如对软盘设备），但是很少。随同 Linux 内核发布的一个文件 Documentation/devices.txt 列举了对块设备和字符设备两种主设备号和次设备号的分配和指定，读者可以参阅。

以前我们还讲过，每项设备都有一个代表着它的设备文件。但是，当一项设备的驱动划分成若干层次而形成一个所谓设备驱动层“堆栈”，尤其是当通过若干个可安装模块实现时，就引发了一个问题：这项设备由几个设备文件代表？是每一子层都有一个文件还是整个“堆栈”只由一个文件代表？我们知道，所谓一个设备文件代表一项设备，实质上是代表一个驱动程序的入口以及与之相联系的数据结构。用“面向目标程序设计”的话来说，就是代表着一个目标。所以，一个具体的子层（模块）是否有相应的设备文件取决于是否构成意义上独立的“设备”。在“可安装模块”一节里所引的例子中，低层模块一方面向其上层登记，另一方面又在文件系统中创建起相应的设备文件节点，就是因为这个模块本身就构成一项独立的设备，应用程序有可能需要绕过其高层直接进行读／写。另一方面，当从一个层次进入下一个层次时，通常意味着一次选择，因此需要加以引导。以上面提到的“虚拟终端”为例，当从虚拟终端层进入“物理终端”层时，就意味着一次选择，或者说“转接”：这个物理终端是接在某个串行口 UART 上的“笨终端”（或者模拟笨终端的计算机）呢？还是接在 VGA 卡上的显示器？我们在块设备驱动一节中看到了，在类似的情况下是由主设备号／次设备号引导的，但那并不是惟一的方法。从上层进入下层的路径既可以临时加以选择（例如根据设备号），也可以预先设置好。这种设置可以在系统（或设备）初始化时进行，也可以通过对高层的 ioctl()操作随时进行，还可以是程序中固有的。在以后几节中，读者将通过阅读代码看到这些技巧的运用。

在这一节中，我们将阅读几个简单字符设备驱动程序的源代码，这些设备都是比较简单的，以后还将阅读虚拟终端驱动程序的代码，那就比较复杂了。由于打开设备文件的过程都大致相同，我们就不再重复，而直接从设备的 file_operations 数据结构开始。

我们要看的第一个字符设备，其实谈不上是“设备”，但是却很常用，那就是“空设备”，即/dev/null。大家知道，应用程序在运行的过程中一般都要通过其预先打开的标准输出通道或标准出错信息通道在终端显示屏上输出一些信息，但是有时候（特别是在批处理中）不宜在显示屏上显示这些信息，又不宜将这些信息重定向到一个磁盘文件中，而要求直接使这些信息流入“下水道”而消失，这时候就可以用/dev/null 来起这个“下水道”的作用。如前所述，这个设备的主设备号为 1。主设备号为 1 的设备其实并不是“设备”，而都是与内存有关，或者在内存中（不必通过外设）就可以提供的功能，所以其符号为 MEM_MAJOR，定义见 include/linux/major.h：

==================== include/linux/major.h 19 19 ====================

```
19   #define MEM_MAJOR  1
```

其 file_operations 结构为 memory_fops，定义见 drivers/char/mem.c：

==================== drivers/char/mem.c 613 615 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static struct file_operations memory_fops = {
	open:		memory_open,	/* just a selector for the real open */
};
```

但是，如前所述，主设备号为 1 的字符设备需要根据次设备号进一步区分具体的设备驱动程序，所以 memory_fops 还不是最终的 file_operations 数据结构，还需要由 memory_open()进一步加以确定和设置，其代码在同一文件（drivers/char/mem.c）中：

==================== drivers/char/mem.c 549 584 ====================

```
static int memory_open(struct inode * inode, struct file * filp)
{
	switch (MINOR(inode->i_rdev)) {
		case 1:
			filp->f_op = &mem_fops;
			break;
		case 2:
			filp->f_op = &kmem_fops;
			break;
		case 3:
			filp->f_op = &null_fops;
			break;
#if !defined(__mc68000__)
		case 4:
			filp->f_op = &port_fops;
			break;
#endif
		case 5:
			filp->f_op = &zero_fops;
			break;
		case 7:
			filp->f_op = &full_fops;
			break;
		case 8:
			filp->f_op = &random_fops;
			break;
		case 9:
			filp->f_op = &urandom_fops;
			break;
		default:
			return -ENXIO;
	}
	if (filp->f_op && filp->f_op->open)
		return filp->f_op->open(inode,filp);
	return 0;
}
```

因为/dev/null 的次设备号为 3，所以其最终的 file_operations 数据结构为 null_fops，仍定义于 drivers/char/mem.c：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/char/mem.c 521 525 ====================

```
static struct file_operations null_fops = {
     llseek:        null_lseek,
     read:          read_null,
     write:         write_null,
};
```

由于这个结构中的函数指针 open 是 NULL，在打开文件时没有任何附加操作。当通过 write()系统调用写这个文件时，相应的驱动函数为 write_null()，这个函数的代码也在 drivers/char/mem.c 中：

==================== drivers/char/mem.c 344 348 ====================

```
static ssize_t write_null(struct file * file, const char * buf,
                size_t count, loff_t *ppos)
{
     return count;
}
```

就是说，它什么也不干，而只是返回 count，假装要求写入的那么多字节都已经写好了，实际的效果就是把要写的内容都丢弃了（读者也许会联想到某些公务员的行为特征）！

那么，通过系统调用 read()的读操作又如何？再看 read_null()的代码：

==================== drivers/char/mem.c 338 342 ====================

```
static ssize_t read_null(struct file * file, char * buf,
               size_t count, loff_t *ppos)
{
     return 0;
}
```

返回 0 表示从这个文件中读了 0 个字节，但是并未到达（永远不会到达）文件的末尾 EOF，即-1。

当然，字符设备的驱动不会都这么简单，但是总的框架是一样的。

8.7 终端设备与汉字信息处理

每个受控制的系统都要有个“控制台”（console），主要用于系统的引导、控制和管理。最原始的控制台是一些开关之类的东西，后来改成了电动打字机（TTY 就是电动打字机的意思），再后来又改成了 CRT 终端，即带显象管的终端，大多是不带智能的所谓“笨终端”。另一方面，系统的用户，特别是在多用户系统中，也需要有个作为人机交互手段的独立的终端设备，一般也都采用 CRT 终端。特别地，用户向系统登录时使用的终端就称为该用户的“控制终端”。在早期的应用中，这些终端一般都是面向文字的，特别是面向拼音文字的（如英文、俄文等），一般都没有图形／图像功能，通常也不要求很强的脱机编辑功能，所以在终端设备中一般不需要有微处理器，或者说不带智能，所以称为“笨终端”。严格说来，纯粹作为控制台使用的设备与作为控制终端使用的设备在功能要求上有所不同，但是相比之下这些不同一般都不大，由此而造成的设备价格上的不同也并不突出，所以人们常常倾向于采用同类的终端设备，而把其中一台保留给“系统管理员”用作控制台。

随着计算机应用的普及和发展，在有些应用中对终端设备有了一些特殊的要求，例如对图形／图

像的支持、对脱机编辑功能的要求等等，而其中最重要的则莫过于对非拼音文字如汉字、日文等等的支持。这些终端设备的功能都比较复杂，所以本身都带有微处理器而成为“智能终端”。80 年代初期人们常常看到的“汉字终端”就是支持汉字输入和显示的智能终端，但是智能终端的出现并不给操作系统内核的设计和实现带来显著的影响。以汉字终端为例，内核只要不排斥 16 位的汉字编码就可以了（一般而言，Unix 内核并不排斥汉字编码）。具体到对汉字的输入、显示、打印和各种处理都是由汉字终端、中文打印机以及各种中文应用软件在操作系统以外提供支持的。那时候，典型的计算机系统形式就是一台“主机”带上许多终端，其中一台是由系统管理员使用的“控制台”。

个人计算机（PC）的出现和发展使情况发生了显著的改变。在 PC 机上，人们通过显示器和键盘（通常还有鼠标器）进行人机交互。这些设备合在一起既用作系统的控制台又是用户的控制终端，而且与“主机”融合在了一起，CPU 的一部分“能力”就用于显示器和键盘的驱动。这样，原来独立存在于终端设备中的一些功能，例如对文字显示的支持，就转移到了系统的内核中。以英文字母的显示为例，在以前采用笨终端的 Unix 系统中，内核只要把代表着字母的编码通过串行口送给控制终端（或控制台）就行了，下面就是具体终端的事了，主机对此既不需关心也鞭长莫及。而现在却不同了，内核可能要管到屏面上的每个像素为止，包括根据具体字母的代码和字体找到代表着该字母的“字模”点阵，然后将该点阵中的每一点都映射到屏面上的一个像素。显然，这就相当于在原先 Unix/Linux 内核对终端设备的驱动程序中又增加了一层，使驱动程序更加复杂，但是同时也使对终端设备的驱动更加灵活。例如，这么一来，对字体的选择、颜色的改变以及字母的放大 / 缩小等等就比较容易实现了。而特别重要的是，对非英文字母以及非拼音文字的支持就基本上可以不依赖于硬件了。但是，另一方面，由显示器和键盘所构成的终端设备并不是 Linux 内核所支持的惟一终端设备。只要在 PC 机的串行口（COM1 / COM2）上插上笨终端（或者用来模拟笨终端的其它 PC 机），就照样可以把运行着 Linux 的 PC 机用于多用户环境。进一步，如果在 PC 机上加上足够的串行接口，就照样可以带上几十个甚至上百个终端而成为名副其实的“主机”。当然，对连在串行口上的终端的驱动不同于对显示器和键盘的驱动。Linux 内核对汉字输入和显示的支持只限于其控制台，即甚示器和键盘。也就是说，如果要在除显示器和键盘以外的用户界面上（终端上）支持汉字，那么这些终端或用户界面本身就得带有智能而支持汉字的输入和显示，例如汉字终端，或者在其它 PC 机上运行的汉字终端仿真软件以及（在 web 环境下）支持汉字的浏览器等。不过，技术发展的趋向是连成网络的 PC 机（或“工作站”），所以实际上已经很少有对汉字终端的需求了。

最后，还有一个问题也是要考虑的，那就是与旧有软件的兼容性。一般而言，在 Unix 上开发的 C 语言程序拿到 Linux 上重新编译以后就应该可以运行，而“Shell script”（Shell 脚本）则应该直接（最多略加修改）就可以执行。这样，就不允许对设备驱动的用户界面作太大的改动，例如不能将由显示器和键盘构成的终端与普通终端通过不同的主设备号加以区分，否则碰到类似“echo please wait >> /dev/tty0”的语句就不能执行了。可见，将 Unix 内核的终端设备驱动加以改造，使之既适合由显示器和终端构成的特殊终端设备，又适合普通的、传统的终端设备，还要考虑到兼容性，确是一项富有挑战性的任务，而对多种文字的支持，则使它更加复杂了。

在迄今为止的计算机发展史上，大部分的技术和应用都首先出现在美国和欧洲，因而最初都是以英文为载体的。英文是拼音文字，而且其字母表又很小，连大小写字母加数字加常用符号全都算上还不到 127 个，所以英文（美国英文）的编码方案 ASCII 是 7 位的，这又正好与初期的 RS232 串行接口每次只能传输 7 位（另一位用作检错）相符。虽然当时在计算机中已经广泛采用以 8 位为一个字节，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在用来存储字符时则固定使最高位为 0。后来，由于考虑到一些特殊符号，如一些数学符号和常用的希腊字母，再说 RS232 串行接口上的传输也变得可靠而不再需要对传输的每个字符都加上奇偶检验，才把 ASCII 编码扩充成 8 位。然而，当要把计算机技术和应用推广（或者销售）到其他国家和地区时，不可避免地会碰到一个问题，那就是文字“本土化”的问题，即对所在国家或地区的文字的支持问题。世界上多数国家都使用拼音文字，尽管其字母表有大有小，但是 8 位的编码可以容纳 256 个代码，一般都够了，而且恰好字节的大小也是 8 位，RS232 串行口的传输也是每次 8 位，所以用一个字节表示一个字符就成了事实上的国际标准。但是，还有些国家和地区使用的是非拼音文字，其中最主要的就是所谓 CJKV，即中文、日文、朝鲜文和越南文。可想而知，对于像中文这样的非拼音文字，要继续以一个字节表示一个字是不可能的，因为一个字节总共才有 256 种不同的编码。但是到底以几个字节表示一个汉字，每个字节中用 7 位还是 8 位，是固定长度还是可变长度，怎样编码，在一个字符串中是否允许混合使用单字节编码和多字节编码，以及怎样在二者之间切换等等，则都是值得研究的课题。事实上，在不同的地区、不同的条件下已经发展起了多种不同的编码方案。

很自然地，将各种文字的编码方案加以标准化和一体化的努力也就应运而生。这里所谓“标准化”是指国际标准或跨国行业标准的制订，而一体化则是指将世界上所有的文字都纳入同一编码方案（以及标准）里面。回顾这方面的历史，下面几个标准是必须提到的：

(1) 美团标准 ASCII。
(2) 国际标准 ISO646。
(3) 国际标准 ISO8859。
(4) 国际标准 ISO2022 和中国国标 7 位 GB2312。
(5) Unix 行业标准 EUC 和中国国标 8 位 GB2312 及 GBK。
(6) 计算机及信息行业标准 Unicode。
(7) 国际标准 ISO10646。

先要说明，我们在这里只是为阅读和理解 Linux 内核代码以及汉化 Linux 内核的需要而介绍一些背景材料，采取的是“带着问题学”的实用主义态度。读者要了解详细的、比较完整的材料应参阅有关专著以及这些标准的文本。

美国的国家标准 ASCII（美国标准信息交换码）公布于 70 年代前期。这个标准把数据交换采用的编码定为 7 位。这一方面是由于当时通过串行接口 RS232 的数据传输还不太可靠，需要对每个代码使用一位作为奇偶校验，而且当时传输的速度也比较低，能够在每个代码中节省下一位就可以使总体的吞吐量增加 10%左右。再说，当时人们的头脑中可能觉得 7 位已是不简单了，在此之前还使用过 5 位和 6 位编码呢。另一方面，当时的人们也觉得超过 7 位是没有必要的，因为 7 位编码的容量是 128，而美国英语中的可打印字符才 94 个（包括空格在内）。至于非打印字符，除了作为控制字符使用外，当时的人们也看不出在数据交换中有多大用处。当时的计算机内部结构可说是五彩续纷，有的厂家采用 8 位，有的厂家采用 10 位，数据的内部表示也由厂家自己定义（IBM 就采用 8 位 EBCDIC 编码），所以除了采用划一的“打印模式”外，确实也看不到有“透明”地传输计算机内部数据的必要性和可行性。随着计算机技术的发展，ASCII 自然地变成了事实上的国际标准。一个标准一旦制订发布，各种软件就会以此为准来设计和开发。这样，当过了若干年以后，原先考虑的因素都已经因为技术的进展而不复存在的时候，一大批以此为基础的软件却已经在那里运行，以后开发的软件只好继续采用 7 位编码以维持兼容，从而形成了本不应存在的壁垒。更有甚者，在有些特殊应用中甚至还从 95 个打印字符中

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

“克扣”了一些字符用作控制目的，使可以“透明”传输的字符个数进一步减少。建立在 64 个可以“透明”传递的可打印字符基础上的电子邮件编码方案 Base64，就是一个典型的例子。

在 ASCII 成为事实上的国际标准以后，国际标准化组织 ISO 通过其 ISO646 加以确认，使其成为正式的国际标准“参考版”之一。同时，又从中提出若干如[、]、{、}、\和|等符号的代码，让字母个数多于 26 个的语言可以用这些符号的代码来表示一些在英语中不存在的字母，如 æ 等等。但是，那已经是 80 年代前期的事了，那时 7 位的传输技术实际上早已过时。正因为这样，ISO 从 80 年代中期开始陆续推出了另一个标准 ISO8859，这个标准采用 8 位传输。ISO8859 分成 10 个部分，为世界上存在的几乎所有拼音文字如何进行 8 位编码都作了规定。例如其第 6 部分，即 ISO8859 第 6 部分（ISO8859-6）是阿拉伯文字的编码，第 7 部分（ISO8859-7）是希腊文字的编码，第 8 部分（ISO8859-8）是希伯来文字的编码。对于拼音文字来说，8 位的编码容量都已经够用了。

但是，ISO8859 并没有解决一体化的问题，同一个 8 位代码对于不同的语言文字就代表着不同的字母，在混合使用多种文字的上下文中要加以切换。另一方面，新标准的制订和采用并不意味着旧的标准就退出了舞台。大量基于旧标准的软件已经在使用中，当要开发新产品的时候，虽然明知道新的标准已经发布，但为了不至失去一大块市场，只好从最坏、最严苛的条件作为出发点，即继续采用 7 位编码，或至少要提供用户一个可选项，让用户根据具体情况来设置到底是采用 7 位编码还是 8 位编码。

这几个标准都只考虑了拼音文字，所以都采用了单字节编码。与此相平行，一些采用非拼音文字的国家和地区也已开始为适合其语言文字的编码制订国家标准或行业标准。其中最早的是日本，在 1978 年初发布了 JIS C 6626 标准，这个标准中的汉字部分后来成为台湾地区制订“大五”编码的借鉴。中国在这方面的工作也开展得很早，1980 年就发布了国标 GB2312。当然，这些标谁都采用了多字节编码，大多是双字节编码。

ISO2022 是第一个支持多字节编码，即非拼音文字的国际标准，发布于 90 年代前期。事实上，这又只不过是对有关国家和地区在这方面的实践的事后追认和兼容并包而已。所以，ISO2022-CN 与中国的 GB2312 很接近，ISO2022-JP 与日本的 JIS C 6626 很接近，而 ISO2022-KR 则与韩国的标准很接近。ISO2022 保留对英文字母的单字节 ASCII 编码，当使用 CJK（中／日／韩）文字时就转入双字节编码，两个字节都采用 7 位，取值范围为 0x21～0x7E，即“可打印字符”的范围。为了保证在不同字符集之间正确切换，ISO2022 定义了一些“ Escape 序列”，称为“标示序列”（designator sequence）。与键盘上的 ESC 键相对应的代码为 0x1b，这并不属于可打印字符的范围，在传输中平时不会用到，所以在计算机技术中与终端或传输线打交道时常常用 ESC 的代码作为控制手段，与其它字符组合成 Escape 序列。此外，还要保证在单字节模式与双字节模式之间正确切换，ISO2022 为此定义了两个控制字符。一个是 0x0e，称为“ SO”（Shift Out），表示转出（常规的）单字节模式；另一个是 0x0f，称为“ SI”（Shift In），表示转入单字节模式。这样，当行文至需要使用非拼音文字时，就先通过 SO 进入双字节模式，再通过 Escape 序列选择具体的字符集，例如当目标为 GB2312 时，其序列为“ 0x1b，0x24，0x29，0x41 ”或“ <Esc>$)A”。然后，当要转回单字节模式时，就在字符串中下一个英文字母之前插入一个 SI 代码即可。

这种技术的好处是效率比较高，从传输的角度看浪费很少，但是从信息处理的角度看却有缺点，因为它是面向过程的，要知道对某一个具体的字节应作何种解释取决于这个字节在传输时处于何种模式，所以对于字符串（甚至整个文件）中的内容只能顺序访问而不能随机访问，因为对每个字节的解

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

读都是与上下文有关的。

另一方面，在某些特殊的应用中，特别是电子邮件的传递中，这套技术还是有问题。前面提到过，电子邮件的传递只保证 64 个可打印字符的透明传输，而 0x0e、0x0f 和 0x1b 显然不属于这个范围。对此，有一个中国留美学者提出了采用另一种序列来实现 ASCII 码与 GB2312 码之间的切换，称为 HZ 编码（RFC 1834）。当要从单字节的 ASCII 码切换到双字节的 GB2312 码时，就插入" ~{"，要回到 ASCII 码时则插入" ~}"。这些字符都属于可打印字符。而" ~}"也不对应于任何汉字的编码，显然，这里的关键是赋予字符" ~"以特殊的解释，所以当在单字节模式下真要使用这个字符时，就要在它前面再添个" ~"字符。读者肯定会想到，这与 C 语言中" \"字符的使用在精神上是一致的。

那么，Linux 的内核是否支持 ISO2022（从而 GB2312）呢？前面讲过，这要分两种情况来考虑。第一种情况是如果采用带有智能的终端设备，即支持 GB2312 的汉字终端，此时只要内核不排斥 GB2312 的编码就行了。第二种情况是当内核需要介入汉字的输入和显示，那又是另一个问题了。

先看第一种情况，Linux 的内核是否排斥 ISO2022 呢？这是个字符串处理的问题，主要关系到字符串的界定、比对，以及一些特殊字符的使用。内核中字符串都是以" \0"结尾的，所以如果在双字节编码中有某个汉字的代码包含了一个 0 字节，那就会被内核误认为是字符串的结尾。可是，ISO2022 中两个字节取值的范围都是 0x21～0x7e，因而不会出现这样的情况。同理，在汉字编码中也不会出现像^C、^D、^Z 以及"退格"这些控制字符，也不会有 0xff（EOF 定义为-1）。再看字符串的比对。内核中要用到字符串比对主要是在文件系统的路径名搜索。在包含汉字的路径名中，需要把换码字符 SI / SO 作为每个节点名字符串的一部分，对于像 Linux 这样对文件名长度并无过分限制的操作系统来说，问题似乎也不大。可是，对另一个特殊字符的解释就成问题了，这个字符是" /"。例如，假定有这样一个路径名：" /usr/students/classes/化工 98"，这就造成问题了。这是因为汉字"化"的编码中含有 0x2f，也就是" /"。读者在文件系统一章中阅读过 path_walk()的代码，在那里这个字符一定会被当作节点名的分隔符而表示另一个层次。这样的汉字当然有很多。所以，结论是：只要不使用汉字作为文件名或目录名，Linux 内核便不会排斥汉字的使用。同样的结论也适用于 Unix。对于 80 年代准备要带上几十个汉字终端的 Unix 系统来说，这个结论已经是足够令人满意的了。

不管是多么权威的标准，只要是束缚了人们的手脚，一旦客观的、物理的限制因素（如硬件的可靠性等）消除以后就一定会被冲破。但是，冲破已经被广泛接受的标准往往要付出不小的代价，这就是新产品的部分市场或适用范围。可要是这市场本来就不存在呢？那自然又作别论。80 年代正是 Unix 气势如虹，从传统的小型机、中型机乃至大型机的手中攻城掠地的时候。当时，硬件（RS232 接口）方面对采用 7 位通信方式的限制已经不存在，而 Unix 机通信的对象一般也都是 Unix 机，不存在已经有大量旧软件在运行而必须保持兼容的问题。在这样的情况下，标准就失去了权威。为了将 Unix 推行到采用非拼音文字的国家和地区，采用了一种称为 EUC（扩充 Unix 编码）的 8 位多样编码方法。EUC 也像 ISO2022 一样混合使用单字节和多字节（一般是双字节）编码。单字节编码也是 7 位的 ASCII 码，也就是说每个字节的最高位永远是 0；但是多字节编码则采用 8 位，把其中某几个字节的最高位设置成 1，因而不使用换码控制字符就可以达到在单字节和多字节两种模式之间的切换。（但是也有例外，EUC 实际上允许在 4 种编码（即文字）之间进行切换，其中 ASCII 为编码 0。从 ASCII 切换到编码 1 可以通过将最高位设成 1 来实现；而若要切换到编码 2 则要插入一个控制字符 0x8e；切换到编码 3 则要插入一个 0x8f。不过在 GB2312 和 GBK 中都不考虑编码 2 和编码 3。）

因目标语言或地区的不同，EUC 编码又分成 EUC-CN（中国）、EUC-JP（日本）和 EUC-KR（韩

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

国）、EUC-TW（中国台湾）4 种。我们在这里只关心 EUC-CN，它的基础仍旧是 GB2312，所以又常常称为“ 8 位国标”或干脆就称为“国标”。它采用单字节和双字节混合编码。单字节编码与 ASCII 相同，字节的最高位永远为 0；而表示汉字的双字节编码用两个字节的最高位均为 1。具体地说，两个字节的取值范围都是 0xa1～0xfe（不包括 0xff）。那么，Unix 的内核会不会排斥这些编码呢？显然不会，因为代表着汉字编码的每个字节的最高位都是 1。而且这意味着甚至可以用汉字作为文件名或目录名了。以前面讲过的“化”字为例，它的第二个字节在 7 位编码中为 0x2f，所以与“ /”冲突，而现在则改成 0xaf 了。

中国后来又公布了对国标的扩充 GBK，也是采用 8 位编码，但是把第一个字节的取值范围扩充成 0x81～0xfe；把第二个字节的取值范围扩充成 0x40～0x7e 和 0x80～0xfe。显然这是为了容纳更多的汉字。虽然 GBK 编码中的第二个字节有可能最高位为 0，但不会造成什么问题，因为在 0x40～0x7e 这个区间并没有什么“敏感字符”。

EUC-CN 和 GBK 比 ISO-2022 显然有改进，但还是有缺点。首先，汉字编码的两个字节最高位均为 1（或均有可能为 1）。使得在传输时容易因误码而使相邻字节的结合发生错位的情况。在这两种编码中，对每个具体字节的解释仍不是与上下文无关的（不过比 ISO2022 采用换码控制字符的编码要好多了），所以还是不能支持对字符串内容的随机访问。其次，EUC 的容量毕竟还是有限，设想如果在同一个文件中既要使用英文，又要使用中文、蒙古义、韩文，还要使用希伯来文，那又该怎么办呢？

上面提到的这些标准都没有考虑（更谈不上解决）世界文字编码一体化的问题。这些编码中没有一种是能够覆盖全世界所有常用文字的。而在受到覆盖的各国文字中，则又常常重复使用相同的代码，例如代码 0x5B 在 ASCII 中表示“ [，而在丹麦文字中却表示“ Æ”，此谓“一码多字”。另一方面，有些文字却又以不同的代码重复地出现于不同的编码中，特别是中文、日文、韩文中都使用汉字，但是同一个汉字在这三种文字编码中的代码却又往往不同，从而造成“一字多码”。还有，单字节与多字节之间的切换，无论用或不用换码控制字符，都在一定程度上使对字符串中具体字节的解释依赖于上下文，从而使得对字符串内容的随机访问变得困难或甚至不可能，这同时也使得字符串在传输过程中的抗干扰能力变差。再说，有些人可能还觉得以单字节代表英语编码，而以多字节代表其它各种文字的编码实际上是把英语置于一种中心地位，因而有失公平。

有这么多的缺点存在，人们寻求世界文字编码一体化的努力就毫不奇怪了，而 Unicode 正是这样一种一体化的编码方案，这是由一个行业协会性质的组织“ Unicode Consortium”发展起来的行业标准。提到 Unicode 就不能不提出另一个国际标准 ISO10646，因为这二者的形成和发展是紧密结合在一起的。最初，在 80 年代前期，ISO 组成了一个工作组 WGZ 从事一体化编码方案的制订，并提出了一个框架性的方案。通常，国际标准从制订到批准、采纳是需要相当长的时间的。另一方面，作为国际标准，其视野自然更为宽阔，其结构也常常更为严谨，很多问题都不是一下子就能定得下来的。可是，当时的工业界却觉得形势逼人，不能再等，于是就组成了 Unicodc Consortium，在 ISO 工作组框架的基础上以一种比较实用的态度制订行业标准 Unicode。后来，ISO 的框架性方案经过多年的发展成为国际标准 ISO10646-1，即 ISO10646 的第一版。目前有 ISO10646-1:1993 和 ISO10646-1:2000 两个文本，分别发表于 1993 年和 2000 年。与此相平行，Unicode Consortium 在 1990 年便发表了 Unicode 1.0 版，以后又分别在 1996 年和 1999 年发表了 2.0 版和 3.0 版。不过，ISO 和 Unicode Consortium 在这方面的工作既互相平行，又密切结合，并且互相靠拢互相融合。1990 年 Unicode Consortium 发表了 Unicode 1.0 版以后，1991 年双方就进行了一融合，从而产生了 1993 年的 ISO10646-1 文本。而 Unicode Consortium 又

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在此基础上发表了 Unicode 的 1.1 版，然后又进一步向 ISO10646 靠拢而在 1996 年发表了 2.0 版。所以，这二者之间的关系是一种互相靠拢、互相补充、且互相融合的良性互动关系。

那么，Unicode 到底是什么样的呢？这二者到底有些什么异同呢？我们在这里只是结合阅读 Linux 内核代码的需要作一简略的介绍，欲知详情的读者可以参考有关专著和文本。在这方面，有两本书是值得推荐的：一本是 Ken Lunde 的 *CJKV Information Processing*；另一本是 Tony Graham 的 *Unicode: A Primer*。

首先，ISO10646 和 Unicode 都意在将全世界所有的文字都包罗在同一个编码方案中，只是 ISO 在这方面更加追求完美，而 Unicode 则采取相对而言比较实用的态度。其次，二者都采用固定长度的字符编码。这样就摆脱了在单字节代码和多字节代码之间来回转换的问题，从而使得对字符串内容的随机访问成为可能。但是，因为 ISO 的“胃口”更大，所以 ISO10646 标准选择采用 4 字节编码 UCS-4；而 Unicode 标准则采用更为紧凑的 2 字节编码 UCS-2。以字母“ a”为例，在 ASCII 编码中其代码为 0x61，而在 Unicode 中为 0x0061，在 ISO10646 中则为 0x00000061。显然，固定长度编码在许多情况下增加了对存储量以及传输带宽的要求（或者说“浪费”了一些存储空间和带宽），不像以前可变长度编码那么紧凑。但是，在计算机系统的存储空间日益加大，成本日益下降，通信能力日益上升的背景下，这一点已经是无关紧要的了。与取得的好处相比，这个代价是值得花的。如果说以前的单字节与多字节混合编码的方案与 ISO10646 的编码方案是两个极端的话，那么 Unicode 就是二者间的折衷。另一方面，虽然 Unicode 的 2 字节编码容量还是太小（例如，还不能把古埃及的文字包罗进去，更不能把中国的甲骨文字包罗进去），但毕竟已经可以覆盖全世界正在积极使用中的所有文字和符号。所以 ISO 和 Unicode Consortium 才在 1991 年对两个标准进行了融合。一方面 ISO 接受 2 字节码的 UCS-2 为 ISO10646 的一个子集，称为“基本多语言平面 ” BMP（Basic Multilingual Plane），并采纳了 Unicode 中的具体代码。另一方面，Unicode Consortium 也认识到 2 字节编码的容量确实还不够，因而在 2 字节编码空间中割下一块（0xD800～0xDFFF）称为“代用码”（surrogate），当 2 字节编码不够用时就可以采用两个“代用码”，即以 4 个字节来表示 0x10000～0x10ffff 区间的代码。那么，“代用码”的采用会不会又破坏了对字符串内容的随机访问呢？不会，因为当采用“代用码”时位于高位的两个字节一定在 0xD800～0xDFFF 范围内，而低位的两个字节一定在 0xDC00～0xDFFF 范围内，这两块代码在正常的 UCS-2 编码中是保留不用的。由两个 2 字节代用码所表达的代码数值按以下公式计算：

$$N = 0x10000 + (H - 0xd800) \times 0x400 + (L - 0xdc00)$$

这里的 H 为高位代用码，其取值范围为 0xd800～0xdbff；L 为低位代用码，其取值范围为 0xdc00～0xdfff。

这么一来，与原先设想的 UCS-2 编码就有所不同了，所以 ISO 进一步将这种编码方案称为 UTF-16，意为过渡性的 16 位编码格式。而前述的 BMP，也就是 Unicode，实际上对应于 UTF-16，而不是原先的 UCS-2。

最后，无论是 ISO10646 或 Unicode，都排除了“一字多码”的情况，从而实现了字符（符号）与代码间的一一对应。目前，Unicode 包罗了 49194 个字符和符号的代码，其中汉字为 27786 个，这些汉字是将中文、日文和韩文中使用的汉字加以整合而成的。通过“代用码”的使用，还可以将这个容量扩大 16 倍。

可想而知，从以单字节为主，以 ASCII 码为基础的字符串处理到一律采用双字节的 Unicode 字符

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

串处理，这是一个重大的转变，而如果还要考虑兼容性的话，那就更是个挑战。不说别的，就拿字符“ a”的 Unicode 代码 0x0061 来说吧，这里面就有一个字节是 0，而传统的字符串就是以 0 结尾的。其它如 0x03（^C）、0x04（^D）、0x1b（ESC）、0x2f（/）等等字节也是随时都有可能碰到，简直是“防不胜防”。（如果可以抛开原来的单字节字符串处理不管，而在 gcc 中将数据类型 char 改成与 unsigned short 等价，那么许多现有的软件只要重新编释一下就可以支持 Unicode 了（那时候，字符串就是以 0x0000 而不是 0x00 结尾的了）。事实上，java 语言正是这样做的，在 java 语言中 char 是 16 位的而不是 8 位的。可是那样一来许多软件（包括 Linux 内核）中一些数据结构的大小就变掉了，所以问题并不简单。）

所以，指望在一个短时期内完成这种转变显然是不现实的。有鉴于此，Unicode Consortium 和 ISO 从一开始就提供了过渡性的方案。第一个过渡性方案称为 UTF-8，是从 8 位到 16 位（即双字节）的过渡方案；第二个称为 UTF-16，实际上就是 Unicode，是从 16 位到 32 位（即 4 字节）的过渡方案。

当然，我们在这里关心的只是 UTF-8，这是一种什么样的编码方案呢？简而言之，那就是单字节与多字节相结合。读者也许马上就会问：怎么绕了半天又回到单字节与多字节相结合了？我们现在的 ISO2022、GB2312、GBK 等等不正是单字节与多字节相结合吗？是的，虽然从原理上讲都是单字节与多字节相结合，但是具体的编码方法不同。UTF-8 的编码规则为：

- 7 位 ASCII 码保持原状不变。也就是数值在 0x00～0x7F 范围内的 Unicode 代码在 UTF-8 中为单字节，并且最高位为 0。
- 不能用 7 位表示，但是可以用 11 位表示，也就是数值在 0x80～0x7FF 范围内的 Unicode 代码在 UTF-8 中为双字节。第一个字节的最高三位为 110，然后是这 11 位中的高 5 位；第二个字节的最高两位为 10，然后是 11 位中的低 6 位。
- 不能用 11 位表示，但可以用 16 位表示的数值，即数值在 0x800～0xFFFF 范围内的 Unicode 代码在 UTF-8 中为三字节。第一个字节的最高四位为 1110，然后是 16 位中的最高 4 位；第二个字节的最高二位为 10，然后是 16 位中的中间 6 位；第三个字节的最高两位也是 10，然后是 16 位中的最小 6 位。
- 对于超出 16 位，但是可以用 21 位表示的数值，即数值在 0x10000～0xlFFFFF 范围内的代码，在 Unicode 中要用两个“代用码”（surrogate）来表示，而在 UTF-8 中则为四字节编码。第一个字节的最高 5 位为 11110，然后是这 21 位中的最高三位；第二、三、四个字节的最高两位都是 10，然后各载有 21 位中的 6 位。

所以，一个字节的最高位为 0 表示这是单字节的 ASCII 码，最高位为 1 表示这是多字节代码中的一个字节，而该代码中的第一个字节的最高两位均为 1，并且连续有几位为 1 就表示该代码中有几个字节。代码中其余字节的最高两位均为 10。

这样的编码方法有什么好处呢？让我们来看看 UTF-8 编码的性质：

(1) 与 ASCII 码兼容，但是在这方面 UTF-8 并没有什么特别的优势，因为其它各种编码也都是与 ASCII 兼容的。

(2) 不使用换码控制字符，多字节代码中每个字节的最高位都是 1，并且每个多字节代码中的第一个字节很容易与其余的字节相区别，因而不存在“配对”是否正确的问题。UTF-8 编码的这个性质使对字符串内容的随机访问成为可能。当从一个字符串中随机选取一个字时，如果该字节的最高位为 0，则为单字节的 ASCII 码；如果最高两位均为 1 则为多字节代码中的第一

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

个字节，否则往回扫描最多三个字节必可发现代码中的第一个字节。

(3) 抗干扰能力强，假定传输过程中某个字节的最高位从 0 变成了 1 或从 1 变成了 0，则受到损坏的最多只是这个字节本身所在的代码加上与它相邻的代码。而不会像在其它编码方案中那样引起成片的损坏（例如，因换码字符损坏以及因“配对”错位所造成的损坏）。

(4) 仍与 Unicode 一样，是一体化的编码。

(5) 凡是能接受 8 位码的软件就不会排斥 UTF-8 编码，因在它的多字节代码中不含任何特殊字符（如 0x00、0x03、0x1b、0x2f 等等）。或者说，所有能接受 8 位编码的软件都是对 UTF-8 透明的。

那么，Linux 内核对 Unicode 和 UTF-8 的接受程度如何呢？我们不妨带着问题来重温一下 path_walk() 的代码。显然，path_walk()的代码对 UTF-8 代码是透明的，因为多字节的 UTF-8 码中不含有如 0x00、0x2f 等等这些特殊字符。所以，凡是 UTF-8 编码的文字，包括汉字，都可以用在文件名和目录名中，更可以用在文件内容的任何字符串中。可是，path_walk()对 Unicode 还是不透明的，所以 Unicode 不能用在文件名和目录名中。在这方面，对 Linux 内核还有工作要做，这也是今后 Linux 内核继续改进的方向之一。那么，退而求其次，以 Unicode 作为文件内容，就像从前的 Unix 系统带上汉字终端那样，是否可以呢？读者在阅读了有关的内核代码以后就会看到，这是可以的。

了解了上面这么些背景材料以后，我们可以回到终端设备的驱动了。

如前所述，在 PC 机上一般总是以显示器和键盘（可能还有鼠标器）的组合作为控制台的，这二者的结合就相当于一个终端。但是，有些情况下一个系统的控制台不止一个，所以 Linux 将同一套物理的显示器和键盘复用于若干“虚拟控制台”（virtual console），让用户通过“Alt”键与功能键“F1”至“F12”的组合来选择将某个虚拟控制台作为系统的当前控制台。由于一般键盘上有 12 个功能键，所以可以有 12 个虚拟控制台（或者虚拟终端），分别对应着设备文件/dev/tty1 至/dev/tty12。系统在初始化以后以/dev/tty1 为当前控制台。此外，/dev/tty0 永远代表着系统的当前控制台，所以如果用户按了 Alt+F2 键，则/dev/tty0 就等价于/dev/tty2。在/dev 目录中还有一个节点/dev/console，一般都是连接到/dev/tty0，所以也代表着系统的当前控制台。

这样做有什么好处呢？一来系统的用户可以在不同的虚拟终端上登录（不过 Linux 只在前面 6 个虚拟终端上创建 login 进程）并启动不同的作业，然后通过 Alt 键和功能键的组合在这些虚拟控制台之间切换，根据需要使得在某个虚拟终端上启动运行的作业成为当前的“前台作业”。另一方面，系统在引导、初始化以及运行中要显示很多信息，如果让这些信息显示在同一个虚拟终端上，则许多信息混在一起不容易看清，并且很快就被后来的信息所覆盖。如果将这些信息分门别类地显示在不同的虚拟终端上，则用户可以有选择地阅读，这样就方便多了。作为字符设备的这些虚拟终端的主设备号为 4，次设备号则与“终端号”相对应，如/dev/tty0 的次设备号为 0，/dev/tty1 的次设备号为 1，等等。

这些设备文件实际上都代表着一个输出缓冲区，写入这些设备的内容都写在缓冲区中，直到选择某个特定虚拟终端时才将其缓冲区的内容显示到显示器屏幕上。从这些设备读就不一样了，从虚拟终端读实际上是从键盘读，但是键盘只属于当前控制台，所以从虚拟终端的（同步）读操作一直要到选择了该虚拟终端作为当前控制台并从键盘输入后才会返回。

主设备号为 4 的字符设备并非全是虚拟终端，还包括了一些通过常规 UART 串行口连接的实际的终端设备。具体讲，除次设备号为 0 的/dev/tty0 代表着当前控制台，叫当前虚拟终端外，次设备号 1～

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

63 分别为/dev/tty1 至/dev/tty63，代表着 63 个虚拟终端（虽然在 PC 机键盘上只有 12 个功能键，因而只能在前 12 个虚拟终端中作出选择）；次设备号 64 至 255 则分别为/dev/ttyS0 至/dev/ttyS191，它们代表着 192 个可能的 UART 串行口，即一般的串行终端设备。

此外，与/dev/tty0 和/dev/tty1 至/dev/tty63 相对应，在/dev 目录下还有一些主设备号为 7、代表虚拟终端缓冲区的“字符设备”文件/dev/vcs 和/dev/vcs1 至/dev/vcs63。这些设备文件与代表虚拟终端的设备文件相对应，但是有所不同。从某个虚拟终端缓冲区（例如/dev/vcs2）读就是从相应虚拟终端的输出缓冲区读，而不是像虚拟终端本身那样是从键盘读。二者的写操作也有所不同，对虚拟终端缓冲区的写操作只是简单的对线性缓冲区的操作，而并不像虚拟终端那样以缓冲区来模拟显示器的屏幕。这两种设备的 file_operations 数据结构也不同，一为 vcs_fops，一为 tty_fops。

不仅如此，在/dev 目录下还有另外 64 个主设备号也是 7 的字符设备/dev/vcsa 和/dev/vcsa1 至/dev/vcsa63，它们的次设备号为 128 至 191。这些字符设备也对应着 63 个虚拟终端的缓冲区，所不同的是这些缓冲区是带有字符“属性”的缓冲区，文件名中的字母“ a”就是“ attribute”的意思。对 VGA、EGA 等图形卡和 BIOS 有所了解的读者也许知道，当这些图形卡工作于字符模式时可以让每个字符带上一个表示属性的字节以控制该字符显示时的颜色、亮度等属性。

读者也许已经觉得这些终端设备不简单了。然而还不止于此，在/dev 目录下还有主设备号为 5 的字符设备。这些设备统称为“替换”（alternate）终端设备。其中次设备号 64～225 分别为/dev/cua0 至/dev/cua191，共 192 个使用串行口的终端设备，称为“ callout”终端设备，与前述的/dev/tttyS0 至/dev/ttyS191 相对应。那么这二者又有什么区别呢？这就要从终端设备与主机的连接方式说起。终端设备与主机（确切地说是主机的串行口）的连接方式大体上有两种，一种是“本地”连接方式，这就是一般“计算中心”里把几十个乃至上百个终端设备直接（一般都是通过 RS232C 电缆）连接到主机上各个串行口的方式。在这种连接方式里，终端设备与主机的连接是静态的，把哪一个终端的电缆（不管有多长）插入哪一个串行口，这个终端在物理上就静态地连接到了这个串行口。终端的电源可以关闭，但是二者物理上的连接却并不改变。另一种是“远程”连接方式，在这种方式里，终端设备与主机并不直接相连，而是各自通过一个 modem 经电话线网络对接，所以终端设备与主机的连接是动态的，不同的终端设备在不同的时间里可以通过电话线网络连接到主机的同一个串行口上。然而，这里就有一个问题，就是由谁来启动这种动态连接呢？一种办法是让主机一方启动，称为“ callout”，此时的串行口工作于“呼叫”（callout）模式，相应的终端设备称为“呼出型”（callout）终端设备，即/dev/cua0 至/dev/cua191。另一种办法是让终端一方通过其 modem 拨号启动，称为“ dial in”，此时的串行口工作于“拨号”模式，相应的终端设备称为“拨入型”终端设备，即/dev/ttyS0 至/dev/ttyS191。显然，这二首的驱动程序是略有不同的。

除这些设备文件以外，还为一些主要的串行接口卡厂商分配了专用的主设备号。例如，对于 Cyclades 的串行卡，就分配了 19 和 20 两个字符设备主设备号。其中 19 用于 dial in，设备文件名为/dev/ttyC0 至/dev/ttyC31；而 20 则用于“ call out”，设备文件名为/dev/cub0 至/dev/cub31。像这样的例子还有很多，在 Linux/Documentation 目录中有一个文件 devices.txt，列出了所有已经分配的设备号，读者可以参考。

上面讲到的这些终端设备都是物理上存在的，即使是“虚拟终端”，最终也要对应到一个物理的终端上。就拿显示器和键盘的组合来说，虽然与传统的终端设备有所不同，但毕竟具备了构成一个终端的物理要素。然而，随着计算机技术的发展，出现了对一种特殊“终端设备”的需求，这种设备在逻

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

辑上是终端设备，可是实际上却不是，所以称为“伪终端”（pseudo tty)。伪终端总是成对地使用的，就好像是一个管道的两端。一端的设备称为“主设备”（master)，其主设备号为 2，设备名为/dev/ptyAX，这里的 A 表示 16 个字母“ pqrstuvwxyzPQRST”中的一个，X 则为 16 个 16 进制数字（0～f）之一，这样一共可以有 256 个伪终端主设备。另一端的设备称为“从设备”（slave)，其主设备号为 3，设备名则为/dev/ttyAX，同样也是 256 个。每一对伪终端设备，例如/dev/ptyp0 和/dev/ttyp0，就好像是通过一个管道连在一起，其“从设备”一端与普通的终端设备没有什么区别，而“主设备”一端则跟管道文件相似。

那么，为什么要有这样的伪终端设备，又为什么要有主设备和从设备配对呢？让我们考虑当一个 Linux（或 Unix）系统采用 X Window 一类的图形用户界面（GUI）时的情况。在这样的系统里，整个显示屏以及键盘都在一个视窗管理进程的控制之下，显示屏上有若干个用来模拟普通终端的窗口，每个这样的窗口都与一个应用进程例如 shell 相联系。但是每个 shell 进程都以为它的标准输入和标准输出（以及标准出错信息）通道都通向一个终端设备，既不知道也无能力控制显示屏上的窗口。怎么办呢？这就要使用伪终端设备了，图 8.7 是个示意图。

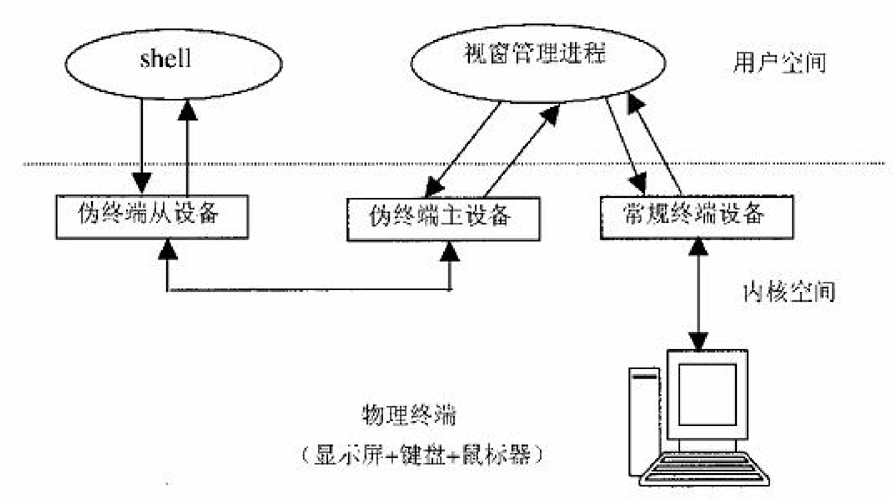


图 8.7　伪终端逻辑示意图

每一对伪终端设备连接着显示屏上的一个窗口和一个应用进程。当视窗管理进程从键盘接收到一个字符时，它先要检查当前的“光标”位置，找出当前“活跃”的窗口和与之对应的伪终端“主设备”（作为一个已打开文件)，然后就把从键盘读入的字符写入到伪终端的“主设备”一端。也就是说，对于键盘输入，视窗管理进程起着中转、搬运的作用，写入伪终端“主设备”一端的字符马上就到达了其“从设备”一端。在那里，对于与“从设备”相联系的进程来说，就跟从普通终端设备读入字符一模一样了。反过来，当“从设备”一边的进程有输出时，它的输出通过伪终端的“从设备”到达“主设备”一端，然后由视窗管理进程读取。视窗管理进程从某个伪终端的“主设备”中接收到字符以后，就要根据具体的打开文件号找到显示屏上相应的窗口，换算成显示屏上的位置，再把接收到的字符在这个位置上显示出来。与伪终端“从设备”相联系的进程，根本就不知道它的终端设备到底是一个物理终端，还是实际上只不过是另一个进程。另一方面，伪终端“主设备”一侧的进程也不一定非得把从伪终端“主设备”接收到的内容在显示屏上的某个窗口里显示出来。例如，在网络环境下“从设备”

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

一端的进程可能仍是 shell，而“主设备”一端的进程则可能是 telnetd，它把从伪终端“主设备”接收到的内容“搬运”到一个网络插口中，反之亦然，这个插口也许通过互联网连接到一千公里以外的另一台机器上。

显然，伪终端设备不同于常规的终端设备，它们的驱动程序也理应有所不同，但是实际上却共享同一个 file_operations 数据结构，即 tty_fops。事实上，主设备号为 2、3、4、5（除 7 以外）的字符设备，以及与分配给各家串行口厂商的主设备号相对应的字符设备，全都使用同一个 file_operations 数据结构。这当然并不意味着所有这些设备都使用相同的驱动程序，而是说明：

(1) 终端设备的驱动程序至少可以分成两个或更多个子层，其中最上层是公共的，所以有相同的入口。

(2) 不同类终端设备的读／写操作不同，例如前述由显示器和键盘构成的控制台终端跟外接的笨终端显然不同，所以一定还有一个类似于 file_operations 那样的函数跳转结构。

(3) 终端设备驱动程序也跟信息传输的方式有关，例如通过 RS232 电缆相连的终端跟通过 Modem 相连的终端肯定不一样。所以，一定还存在着另一个与传输方式有关的函数跳转结构。

事实正是这样，除 file_operations 结构以外，每个终端设备还跟另两个数据结构相联系，一个是 tty_driver 数据结构，定义于 include/linux/tty_driver.h：

==================== include/linux/tty_driver.h 120 179 ====================

```
struct tty_driver {
	int	magic;		/* magic number for this structure */
	const char	*driver_name;
	const char	*name;
	int	name_base;	/* offset of printed name */
	short	major;		/* major device number */
	short	minor_start;	/* start of minor device number*/
	short	num;		/* number of devices */
	short	type;		/* type of tty driver */
	short	subtype;	/* subtype of tty driver */
	struct termios init_termios; /* Initial termios */
	int	flags;		/* tty driver flags */
	int	*refcount;	/* for loadable tty drivers */
	struct proc_dir_entry *proc_entry; /* /proc fs entry */
	struct tty_driver *other; /* only used for the PTY driver */

	/*
	 * Pointer to the tty data structures
	 */
	struct tty_struct **table;
	struct termios **termios;
	struct termios **termios_locked;
	void *driver_state;	/* only used for the PTY driver */

	/*
	 * Interface routines from the upper tty layer to the tty
	 * driver.
	 */
```

```
        int  (*open)(struct tty_struct * tty, struct file * filp);
        void (*close)(struct tty_struct * tty, struct file * filp);
        int  (*write)(struct tty_struct * tty, int from_user,
                  const unsigned char *buf, int count);
        void (*put_char)(struct tty_struct *tty, unsigned char ch);
        void (*flush_chars)(struct tty_struct *tty);
        int  (*write_room)(struct tty_struct *tty);
        int  (*chars_in_buffer)(struct tty_struct *tty);
        int  (*ioctl)(struct tty_struct *tty, struct file * file,
                unsigned int cmd, unsigned long arg);
        void (*set_termios)(struct tty_struct *tty, struct termios * old);
        void (*throttle)(struct tty_struct * tty);
        void (*unthrottle)(struct tty_struct * tty);
        void (*stop)(struct tty_struct *tty);
        void (*start)(struct tty_struct *tty);
        void (*hangup)(struct tty_struct *tty);
        void (*break_ctl)(struct tty_struct *tty, int state);
        void (*flush_buffer)(struct tty_struct *tty);
        void (*set_ldisc)(struct tty_struct *tty);
        void (*wait_until_sent)(struct tty_struct *tty, int timeout);
        void (*send_xchar)(struct tty_struct *tty, char ch);
        int (*read_proc)(char *page, char **start, off_t off,
                   int count, int *eof, void *data);
        int (*write_proc)(struct file *file, const char *buffer,
                   unsigned long count, void *data);

        /*
         * linked list pointers
         */
        struct tty_driver *next;
        struct tty_driver *prev;
};
```

可见，结构中给定了该种终端设备的主设备号以及次设备号的范围，并提供了许多函数指针。这样，对于不同种类的终端设备就有不同的 tty_driver 数据结构，例如控制台终端的 tty_driver 数据结构就是 console_driver。

另一个是 tty_ldisc 数据结构，定义于 include/linux/tty_ldisc.h 中：

==================== include/linux/tty_ldisc.h 103 132 ====================

```
struct tty_ldisc {
        int  magic;
        char *name;
        int  num;
        int  flags;
        /*
         * The following routines are called from above.
         */
        int  (*open)(struct tty_struct *);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
112       void (*close)(struct tty_struct *);
113       void (*flush_buffer)(struct tty_struct *tty);
114       ssize_t  (*chars_in_buffer)(struct tty_struct *tty);
115       ssize_t  (*read)(struct tty_struct * tty, struct file * file,
116                 unsigned char * buf, size_t nr);
117       ssize_t  (*write)(struct tty_struct * tty, struct file * file,
118                  const unsigned char * buf, size_t nr);
119       int  (*ioctl)(struct tty_struct * tty, struct file * file,
120                  unsigned int cmd, unsigned long arg);
121       void (*set_termios)(struct tty_struct *tty, struct termios * old);
122       unsigned int (*poll)(struct tty_struct *, struct file *,
123                      struct poll_table_struct *);
124
125       /*
126        * The following routines are called from below.
127        */
128       void (*receive_buf)(struct tty_struct *, const unsigned char *cp,
129                        char *fp, int count);
130       int  (*receive_room)(struct tty_struct *);
131       void (*write_wakeup)(struct tty_struct *);
132  };
```

结构名中的“ ldisc”应为“ Line Discipline”的缩写，表示“链路规则”的意思。与 file_operations 不同的是，这个结构中不但有供上层调用的函数指针如 open、read、write 等等，还有供下层往上调用的函数指针 receive_buf、receive_room 以及 write_wakeup。此外，结构中还有几个并非函数指针的字段。内核中有个 tty_ldisc 结构数组，用于各种不同的链路规则，包括实际上并不使用链路的“链路规则”，定义于 drivers/char/tty_io.c：

==================== drivers/char/tty_io.c 119 119 ====================

```
119  struct tty_ldisc ldiscs[NR_LDISCS];  /* line disc dispatch table */
```

其大小为 NR_LDISCS，实际上定义为 16，系统在初始化或安装某种驱动模块时通过函数 tty_register_ldisc()将有关的 tty_ldisc 结构“登记”到这个数组中。

如果把具体终端设备的驱动称为“终端设备层”的话，那么 tty_driver 结构是它与其上层，即常规的设备驱动层之间的界面，而 tty_ldisc 结构则是它与其下层，即物理设备层之间的界面。所谓“链路规则”，实际上就是怎样驱动具体的物理接口。

下面，我们通过打开文件操作来看根据终端的种类转接到具体 tty_ldisc 结构的过程。

如上所述，数据结构 tty_fops 是几乎所有终端设备驱动程序的总枢纽，有着特殊的重要性。这个数据结构是在 drivers/char/tty_io.c 中定义的：

==================== drivers/char/tty_io.c 407 416 ====================

```
407  static struct file_operations tty_fops = {
408       llseek:        tty_lseek,
409       read:          tty_read,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
410       write:         tty_write,
411       poll:          tty_poll,
412       ioctl:         tty_ioctl,
413       open:          tty_open,
414       release:  tty_release,
415       fasync:        tty_fasync,
416  };
```

用于打开文件操作的函数是 tty_open()，因为比较长，我们分段阅读（drivers/char/tty_io.c）。

==================== drivers/char/tty_io.c 1285 1360 ====================

```
1285 static int tty_open(struct inode * inode, struct file * filp)
1286 {
1287      struct tty_struct *tty;
1288      int noctty, retval;
1289      kdev_t device;
1290      unsigned short saved_flags;
1291      char buf[64];
1292 
1293      saved_flags = filp->f_flags;
1294 retry_open:
1295      noctty = filp->f_flags & O_NOCTTY;
1296      device = inode->i_rdev;
1297      if (device == TTY_DEV) {
1298           if (!current->tty)
1299                return -ENXIO;
1300           device = current->tty->device;
1301           filp->f_flags |= O_NONBLOCK; /* Don't let /dev/tty block */
1302           /* noctty = 1; */
1303      }
1304 #ifdef CONFIG_VT
1305      if (device == CONSOLE_DEV) {
1306           extern int fg_console;
1307           device = MKDEV(TTY_MAJOR, fg_console + 1);
1308           noctty = 1;
1309      }
1310 #endif
1311      if (device == SYSCONS_DEV) {
1312           struct console *c = console_drivers;
1313           while(c && !c->device)
1314                c = c->next;
1315           if (!c)
1316                      return -ENODEV;
1317               device = c->device(c);
1318           filp->f_flags |= O_NONBLOCK; /* Don't let /dev/console block */
1319           noctty = 1;
1320      }
1321 
1322      if (device == PTMX_DEV) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#ifdef CONFIG_UNIX98_PTYS

		/* find a free pty. */
		int major, minor;
		struct tty_driver *driver;

		/* find a device that is not in use. */
		retval = -1;
		for ( major = 0 ; major < UNIX98_NR_MAJORS ; major++ ) {
			driver = &ptm_driver[major];
			for (minor = driver->minor_start ;
			     minor < driver->minor_start + driver->num ;
			     minor++) {
				device = MKDEV(driver->major, minor);
				if (!init_dev(device, &tty)) goto ptmx_found; /* ok! */
			}
		}
		return -EIO; /* no free ptys */
	ptmx_found:
		set_bit(TTY_PTY_LOCK, &tty->flags); /* LOCK THE SLAVE */
		minor -= driver->minor_start;
		devpts_pty_new(driver->other->name_base + minor, MKDEV(driver->other->major, minor +
driver->other->minor_start));
		tty_register_devfs(&pts_driver[major], DEVFS_FL_NO_PERSISTENCE,
				   pts_driver[major].minor_start + minor);
		noctty = 1;
		goto init_dev_done;

#else   /* CONFIG_UNIX_98_PTYS */

		return -ENODEV;

#endif  /* CONFIG_UNIX_98_PTYS */
	}

	retval = init_dev(device, &tty);
	if (retval)
		return retval;

```

找到代表着目标设备的文件节点以后，其 inode 结构中的 i_rdev 字段就是目标设备的设备号。首先要看这是不是当前进程的控制终端，常数 TTY_DEV 定义于 drivers/char/tty_io.c：

==================== drivers/char/tty_io.c 108 108 ====================

```
#define TTY_DEV MKDEV(TTYAUX_MAJOR,0)
```

就是说，主设备号为 5，次设备号为 0，这就是/dev/tty，表示当前进程的“控制终端”，而并不指向任何物理意义上的终端。一个进程通常都有个“控制终端”。如果没有，那么打开的第一个终端设备一般就成为其控制终端，但是那得要使用表示具体终端的设备文件名才行。有时候虽然一个进程尚无

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

控制终端，并且又要打开一个终端设备，可是却并不要将它作为控制终端，在这种情况下就可以在系统调用 open()的参数 flags 中将一个标志 O_NOCTTY 设成 1，这里的 1295 行将这个标志位的值保存在一个局部量 noctty 中备用。另一方面，每个进程的 task_structure 数据结构中有个指针 tty，指向代表着其当前控制终端的 tty_struet 数据结构。如果这个指针为 0，那就说明这个进程还没有控制终端，所以返回出错代码-ENXIO，表示没有这么个设备。如果要打开的确实是当前进程的“控制终端”，那就已经不是第一次打开了，这里把通过/dev/tty 建立的连接强制设成 O_NONBLOCK。

编译选择项 CONFIG_VT 表示系统中是否配备了前述由显示器和键盘构成的虚拟终端。常数 CONSOLE_DEV 和 SYSCON_DEV 的定义为（见 drivers/char/n_tty.c）：

==================== drivers/char/n_tty.c 48 49 ====================

```
#define CONSOLE_DEV MKDEV(TTY_MAJOR,0)
#define SYSCON_DEV  MKDEV(TTYAUX_MAJOR,1)
```

就是说，如果主设备号为 4 而次设备号为 0，即/dev/tty0（表示当前虚拟控制台），则将其替换成具体的设备号。在支持虚拟控制台的内核中有个全局量 fg_console，表示当前的“前台控制台”。这个变量的数值是从 0 开始的，而各个具体虚拟控制台的次设备号却从 1 开始，所以二者在数值上差 1。这样，1307 行就根据 fg_console 的数值还原了当前虚拟控制台的设备号。另一方面，由于这个虚拟控制台原来就已打开，所以把 noctty 设成 1，保持其原状不变。主设备号为 5 而次设备号为 1 即/dev/console，用于外接的控制台。我们在前面曾经提到过，/dev/console 一般是连接到/dev/tty0 的。但是，在不同的版本里情况有所不同。如果/dev/console 不是连接到/dev/tty0，而是主设备号为 5，次设备号为 1，那么内核在初始化过程中安装模块时会通过一个函数 register_console()登记用作控制台的终端设备，把一个 console 数据结构挂到内核中的 console_drivers 队列中，此后内核便会将需要显示的出错信息（通过 printk()显示的信息）输出到所登记的设备上。我们在后面要讲到 console 数据结构，但是在这里只要知道这个数据结构里有个字段 device，就是具体终端设备的设备号就行了。所以，如果要打开的终端设备是这样的/dev/console，就要在 console_drivers 队列中找到第一个设备号不为 0 的 console 结构，它所提供的设备号就是当前系统控制台的设备号。

最后，常数 PTMX_DEV 的定义见 drivers/char/tty_io.c：

==================== drivers/char/tty_io.c 110 110 ====================

```
#define PTMX_DEV MKDEV(TTYAUX_MAJOR,2)
```

在 Documents/device.txt 文件中可以查到，主设备号为 5 而次设备号为 2 的字符设备文件是/dev/ptmx，是用于伪终端主设备的。我们在前面已经介绍过伪终端设备，其主设备和从设备的主设备号分别为 2 和 3，并且因此而可以有 256 对伪终端设备（次设备号为 8 位）。这实际上是 BSD 版本的做法，现在也一直在沿用，但是 System V 采用了一种不同的方法。在 System V 中设置了一个总的伪终端主设备文件/dev/ptmx，但是它的作用与上述/dev/console 相似，只是提供了一个总的入口，而实际的设备号要在打开文件时临时分配，这样就可以有更多的设备号用于伪终端设备。在早期的应用中，有上百对的伪终端设备已经够用了，所以 System V 这种方法的优点并不明显，但是，随着网络技术和应用的发展，后者的优越性却显得突出起来。用 Unix/Linux 系统作为网络服务器时，可能会要求数百个甚至上千个连接，这时候 256 个伪终端主设备号就不够用了，而采用 System V 的方法就不受这个限制。当然，采用 BSD 的方法也可以通过增加用于伪终端设备的主设备号个数来扩充它的容量，但是那样在

应用程序中就要考虑对大量伪终端设备文件的调度和管理。例如，你怎么知道/dev/ptyr32 已经在使用中，而/dev/ptyx87 还空闲着呢？而 System V 的方法则实际上把这部分工作移到了内核中，效率当然要高一些，并且应用程序也可以简化。所以，在新的 Unix 版本，即 Unix98 中采用了 System V 的方案并加以配套完善，并且又将主设备号 125～135 分配用于伪终端主设备，136～143 用于次设备，这样理论上的容量就可以达到 8×256，即 2048 对伪终端设备。对于 Linux 内核，这是作为一个可选项提供的，相应的条件编译选择项就是 CONFIG_UNIX98_PTYS，不过这需要 2.1 版或以上的 C 程序库 glibc 配套（glibc 为用户程序提供若干库函数，例如 ptsname()返回与一个已经打开的伪终端主设备相对应的次设备文件名）。如果选择了这个可选项，那么内核在初始化时根据实际用于伪终端主设备号的个数（由常数 UNIX98_NR_MAJORS 决定）初始化两个 tty_driver 结构（见下）数组 ptm_driver[]和 pts_driver[]，分别用于主设备和从设备，然后在用户进程通过/dev/ptmx 打开一个伪终端主设备时再动态加以分配。代码中第 1331 行开始的 for 循环扫描 ptm_driver[]中的每个元素，即每个主设备号，并且对于每个主设备号依次尝试各个次设备号，如果以某个主设备号和次设备号的组合调用 init_dev()成功，则一个新的伪终端主设备就打开成功了。但是，接着还要转到 ptmx_found 处为进一步打开次设备作好准备。数组 ptm_driver[]和 pts_driver[]中的每个元素都是一个 tty_driver 数据结构，结构中有个指针 other 用来指向与之配对的另一方，所以 1344 行的 driver 指向 ptm_driver[]中的某个元素，而 driver->other 则指向 pts_driver[]中与之配对的元素。此外，函数 devpts_pty_new()为相应的次设备在内存中创建一个 inode 数据结构，tty_register_devfs 则为之创建一个 devfs 设备节点，例如/dev/pts/0，/dev/pts/134，等等。注意，终端的号码是连续编号的。由于篇幅所限，我们在这里就不深入到这两个函数中去了，有需要或有兴趣的读者可以自行阅读。

除对/dev/ptmx 作特殊处理外，对其它终端设备（包括伪终端设备）的打开文件操作在取得了最终的设备号以后都在 1357 行调用 init_dev()，为需要打开的终端设备建立一个（或找到其）tty_struct 数据结构。

每个已打开的终端设备都由一个 tty_struct 数据结构代表，这种数据结构定义于 include/linux/tty.h：

==================== include/linux/tty.h 245 310 ====================

```
/*
 * Where all of the state associated with a tty is kept while the tty
 * is open.  Since the termios state should be kept even if the tty
 * has been closed --- for things like the baud rate, etc --- it is
 * not stored here, but rather a pointer to the real state is stored
 * here.  Possible the winsize structure should have the same
 * treatment, but (1) the default 80x24 is usually right and (2) it's
 * most often used by a windowing system, which will set the correct
 * size each time the window is created or resized anyway.
 * IMPORTANT: since this structure is dynamically allocated, it must
 * be no larger than 4096 bytes.  Changing TTY_FLIPBUF_SIZE will change
 * the size of this structure, and it needs to be done with care.
 *                             - TYT, 9/14/92
 */
struct tty_struct {
     int  magic;
     struct tty_driver driver;
```

```
    struct tty_ldisc ldisc;
    struct termios *termios, *termios_locked;
    int pgrp;
    int session;
    kdev_t   device;
    unsigned long flags;
    int count;
    struct winsize winsize;
    unsigned char stopped:1, hw_stopped:1, flow_stopped:1, packet:1;
    unsigned char low_latency:1, warned:1;
    unsigned char ctrl_status;

    struct tty_struct *link;
    struct fasync_struct *fasync;
    struct tty_flip_buffer flip;
    int max_flip_cnt;
    int alt_speed;         /* For magic substitution of 38400 bps */
    wait_queue_head_t write_wait;
    wait_queue_head_t read_wait;
    struct tq_struct tq_hangup;
    void *disc_data;
    void *driver_data;
    struct list_head tty_files;

#define N_TTY_BUF_SIZE 4096

    /*
     * The following is data for the N_TTY line discipline.  For
     * historical reasons, this is included in the tty structure.
     */
    unsigned int column;
    unsigned char lnext:1, erasing:1, raw:1, real_raw:1, icanon:1;
    unsigned char closing:1;
    unsigned short minimum_to_wake;
    unsigned overrun_time;
    int num_overrun;
    unsigned long process_char_map[256/(8*sizeof(unsigned long))];
    char *read_buf;
    int read_head;
    int read_tail;
    int read_cnt;
    unsigned long read_flags[N_TTY_BUF_SIZE/(8*sizeof(unsigned long))];
    int canon_data;
    unsigned long canon_head;
    unsigned int canon_column;
    struct semaphore atomic_read;
    struct semaphore atomic_write;
    spinlock_t read_lock;
};
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里先介绍一下其中几个特别重要的成分，结构中其它字段的作用随着代码的阅读自会变得清楚起来。首先是一个 tty_driver 数据结构（261 行）。如前所述，每种终端设备都有自己的 tty_driver 结构，结构中主要是一些函数指针，确定了对具体终端类型的一整套操作。内核中有个链表 tty_drivers，系统在初始化时，或安装某种终端设备的驱动模块时，通过函数 tty_register_driver()将各种终端设备的 tty_driver 结构登记到这个链表中。每当新打开一个终端设备时，就要根据其设备号通过函数 get_tty_driver()在这个链表中找到相应的 tty_driver 结构，并把它复制到具体的 tty_struct 结构中。第二个重要成分就是 tty_ldisc 数据结构（262 行）。如前所述，内核中有个 tty_ldisc 结构数组 ldiscs[]，当新创建一个 tty_struct 结构时，就从该数组中把相应的 tty_ldisc 结构复制到 tty_struct 结构中的这个成分中。第三个重要成分是指针 termios，指向一个 termios 结构。每个逻辑意义上的终端设备接口，如串行口、图形卡和键盘的组合乃至伪终端设备的两端，都有一个 termios 数据结构。这个数据结构在某种程度上可以看作是对 tty_ldisc 结构的补充，它规定了对接口上输入和输出的每个字符所作的处理以及传输的速度，即“波特率”。数据结构的定义在 include/asm-i386/termbits.h 中：

==================== include/asm-i386/termbits.h 10 18 ====================

```
#define NCCS 19
struct termios {
     tcflag_t c_iflag;        /* input mode flags */
     tcflag_t c_oflag;        /* output mode flags */
     tcflag_t c_cflag;        /* control mode flags */
     tcflag_t c_lflag;        /* local mode flags */
     cc_t c_line;             /* line discipline */
     cc_t c_cc[NCCS];         /* control characters */
};
```

同一文件中定义了用于这个结构中各个字段的许多常数（大多是标志位），例如用于 c_lflag 字段的标志位 ISG 为 0000001，ICANON 为 0000002，ECHO 为 0000010。如果将 c_lflag 字符中的这些标志位清成 0。则相应终端设备就工作于所谓“原始模式”（raw mode），否则就工作于所谓“加工模式”（cooked mode）。有关详情在阅读有关代码时还会讲到。限于篇幅，就不在这里列出这些常数的定义了。在 tty_struct 结构中还有个重要的成分 flip，是个 tty_flip_buffer 数据结构。它是终端设备的输入缓冲区，底层的中断服务程序将接收到的字符存储于这个缓冲区中，供其上层读取。这就是我们以前提到过的上层的同步读操作与底层的异步读操作之间的交汇点，其定义在 include/linux/tty.h 中：

==================== include/linux/tty.h 132 149 ====================

```
/*
 * This is the flip buffer used for the tty driver.  The buffer is
 * located in the tty structure, and is used as a high speed interface
 * between the tty driver and the tty line discipline.
 */
#define TTY_FLIPBUF_SIZE 512

struct tty_flip_buffer {
     struct tq_struct tqueue;
     struct semaphore pty_sem;
     char      *char_buf_ptr;
```

```
    unsigned char *flag_buf_ptr;
    int       count;
    int       buf_num;
    unsigned char char_buf[2*TTY_FLIPBUF_SIZE];
    char      flag_buf[2*TTY_FLIPBUF_SIZE];
    unsigned char slop[4]; /* N.B. bug overwrites buffer by 1 */
};
```

与 flip 相应的另一个成分是字符位图 process_char_map，这个位图中的每一位都对应着一个 8 位字符，所以位图的大小为 32 字节。如果某个字符在这个位图中的对应位为 1，就表示可能要对这个字符作出一些特殊的处理或反应（往往只是在“加工模式”下才起作用）。

随着代码的阅读，数据结构中其它字段的作用会慢慢变得清楚起来。函数 init_dev()的代码在 drivers/char/tty_io.c 中，这也是个比较长的函数，也得要分段阅读。

==================== drivers/char/tty_io.c 806 867 ====================

[tty_open()>init_dev()]

```
static int init_dev(kdev_t device, struct tty_struct **ret_tty)
{
    struct tty_struct *tty, *o_tty;
    struct termios *tp, **tp_loc, *o_tp, **o_tp_loc;
    struct termios *ltp, **ltp_loc, *o_ltp, **o_ltp_loc;
    struct tty_driver *driver;
    int retval=0;
    int idx;

    driver = get_tty_driver(device);
    if (!driver)
        return -ENODEV;

    idx = MINOR(device) - driver->minor_start;

    /*
     * Check whether we need to acquire the tty semaphore to avoid
     * race conditions.  For now, play it safe.
     */
    down_tty_sem(idx);

    /* check whether we're reopening an existing tty */
    tty = driver->table[idx];
    if (tty) goto fast_track;

    /*
     * First time open is complex, especially for PTY devices.
     * This code guarantees that either everything succeeds and the
     * TTY is ready for operation, or else the table slots are vacated
     * and the allocated memory released.  (Except that the termios
     * and locked termios may be retained.)
     */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
838
839         o_tty = NULL;
840         tp = o_tp = NULL;
841         ltp = o_ltp = NULL;
842
843         tty = alloc_tty_struct();
844         if(!tty)
845              goto fail_no_mem;
846         initialize_tty_struct(tty);
847         tty->device = device;
848         tty->driver = *driver;
849
850         tp_loc = &driver->termios[idx];
851         if (!*tp_loc) {
852              tp = (struct termios *) kmalloc(sizeof(struct termios),
853                                    GFP_KERNEL);
854              if (!tp)
855                   goto free_mem_out;
856              *tp = driver->init_termios;
857         }
858
859         ltp_loc = &driver->termios_locked[idx];
860         if (!*ltp_loc) {
861              ltp = (struct termios *) kmalloc(sizeof(struct termios),
862                                     GFP_KERNEL);
863              if (!ltp)
864                   goto free_mem_out;
865              memset(ltp, 0, sizeof(struct termios));
866         }
867
```

调用参数有两个。一个是设备号，由于所有的终端设备部共用同一个 file_operations 结构，从而同一个函数 tty_open()，这里需要进一步根据设备号来区分具体的设备和操作。另一个参数是双重指针，指向一个 tty_struct 指针，目的是用来返回一个 tty_struct 数据结构。

简言之，init_dev()的任务就是根据设备号找到或创建目标终端设备的 tty_struct 数据结构，并且执行在打开该设备时所需的附加操作。首先是通过 get_tty_driver()根据设备号从 tty_drivers 链表中找到相应的 tty_driver 数据结构（815 行），这个结构中有个指针 table，指向一个 tty_struct 结构指针数组。数组中包含了所有该种终端设备的 tty_struct 结构指针。举例来说。主设备号为 5 的“辅助 TTY 设备”实际上包括了好几种不同的终端设备，其中次设备号 64～255 代表着 192 个“呼出型”串行口终端设备，设备文件为/dev/cua0 至/dev/cua191。所以，在链表 tty_drivers 中就有一个代表着呼出型终端设备的 tty_driver 数据结构，而这个结构中的指针就指向一个大小为 192 的 tty_struct 结构指针数组。由于这种终端设备的次设备号不是从 0 开始的，所以要有一个 minor_start 字段，用以说明其起始次设备号，而具体的次设备号与起始次设备的差就用作访问该数组的下标（见 819 行）。如果一个终端设备已经打开，即其 tty_struct 结构已经存在，那么数组中相应的指针就不为 0，那就不需要创建了（见 828 和 829 行），否则就要通过 alloc_tty_struct()分配空间，然后就是对这个数据结构的初始化。注意 848 行是数据结构

的赋值，是把整个 tty_driver 数据结构复制到目标设备的 tty_struct 结构的内部。这样一来，前述两个函数跳转结构之一就与目标设备挂上了钩。函数 initialize_tty_struct()的代码也在 drivers/char/tty_io.c 中：

==================== drivers/char/tty_io.c 1958 1977 ====================

[tty_open()>init_dev()>initialize_tty_struct()]

```
1958 static void initialize_tty_struct(struct tty_struct *tty)
1959 {
1960     memset(tty, 0, sizeof(struct tty_struct));
1961     tty->magic = TTY_MAGIC;
1962     tty->ldisc = ldiscs[N_TTY];
1963     tty->pgrp = -1;
1964     tty->flip.char_buf_ptr = tty->flip.char_buf;
1965     tty->flip.flag_buf_ptr = tty->flip.flag_buf;
1966     tty->flip.tqueue.routine = flush_to_ldisc;
1967     tty->flip.tqueue.data = tty;
1968     init_MUTEX(&tty->flip.pty_sem);
1969     init_waitqueue_head(&tty->write_wait);
1970     init_waitqueue_head(&tty->read_wait);
1971     tty->tq_hangup.routine = do_tty_hangup;
1972     tty->tq_hangup.data = tty;
1973     sema_init(&tty->atomic_read, 1);
1974     sema_init(&tty->atomic_write, 1);
1975     spin_lock_init(&tty->read_lock);
1976     INIT_LIST_HEAD(&tty->tty_files);
1977 }
```

这里 1962 行又是数据结构的赋值，这一次是把整个 tty_ldisc 数据结构复制到了目标设备的 tty_struct 结构的内部。这样，前述的两个函数跳转结构就都与目标设备挂上了钩。不管是什么种类的终端设备，其默认的 tty_ldisc 数据结构都是 ldiscs[N_TTY]，这里 N_TTY 定义为 0。这就是虚拟终端，即 VGA 卡加键盘所采用的链路规则，这种"链路"的规则就是不存在物理的链路。至于其它终端设备，则可以在以后通过系统调用 ioctl()设置采用特殊的链路规则。

回到 init_dev()的代码中。如前所述，在 tty_driver 结构中有个指针 termios 指向一个 termios 指针数组，这个数组也是以终端号（次设备号减去该类终端设备的起始次设备号）为下标的。与此相平行，还有个指针 termios_locked 指向另一个 termios 结构指针数组。如果这两个数组中对应于正欲打开的终端设备的 termios 结构指针为 0，就表示尚未为之创建 termios 数据结构，所以要为之分配空间（见 852 行和 861 行）并初始化。其中属于 termios[]的数据结构从 driver 结构中的一个"样板"termios 结构 init_termios 复制（见 856 行），而属于 termios_locked[]的数据结构则初始化成全 0（见 865 行）。

再往下看（drivers/char/tty_io.c）：

==================== drivers/char/tty_io.c 868 913 ====================

[tty_open()>init_dev()]

```
    if (driver->type == TTY_DRIVER_TYPE_PTY) {
        o_tty = alloc_tty_struct();
        if (!o_tty)
            goto free_mem_out;
        initialize_tty_struct(o_tty);
```

```
873         o_tty->device = (kdev_t) MKDEV(driver->other->major,
874                     driver->other->minor_start + idx);
875         o_tty->driver = *driver->other;
876
877         o_tp_loc  = &driver->other->termios[idx];
878         if (!*o_tp_loc) {
879             o_tp = (struct termios *)
880                 kmalloc(sizeof(struct termios), GFP_KERNEL);
881             if (!o_tp)
882                 goto free_mem_out;
883             *o_tp = driver->other->init_termios;
884         }
885
886         o_ltp_loc = &driver->other->termios_locked[idx];
887         if (!*o_ltp_loc) {
888             o_ltp = (struct termios *)
889                 kmalloc(sizeof(struct termios), GFP_KERNEL);
890             if (!o_ltp)
891                 goto free_mem_out;
892             memset(o_ltp, 0, sizeof(struct termios));
893         }
894
895         /*
896          * Everything allocated ... set up the o_tty structure.
897          */
898         driver->other->table[idx] = o_tty;
899         if (!*o_tp_loc)
900             *o_tp_loc = o_tp;
901         if (!*o_ltp_loc)
902             *o_ltp_loc = o_ltp;
903         o_tty->termios = *o_tp_loc;
904         o_tty->termios_locked = *o_ltp_loc;
905         (*driver->other->refcount)++;
906         if (driver->subtype == PTY_TYPE_MASTER)
907             o_tty->count++;
908
909         /* Establish the links in both directions */
910         tty->link   = o_tty;
911         o_tty->link = tty;
912     }
913
```

对于打开文件操作，伪终端是有其特殊性的，所以如果要打开的是伪终端设备（不论是主设备或次设备）就要进行一些特殊的处理。打开伪终端设备时，tty_struct 结构是成对地分配、创建的，这样才能使两个数据结构预先“背靠背”地互相挂上钩。以打开一个伪终端主设备为例，在创建了主设备一侧的 tty_struct 结构 tty（见前面的 843 行）以后，还要创建从设备一侧的 o_tty（见 869 行）。这里 o_tty 显然是表示“ the other tty”（另一个 tty）的意思。伪终端主设备和从设备的 tty_driver 数据结构都通过指针 other 指向对方，所以如果 875 行中的 driver 指向主设备的 tty_driver 结构，则 driver->other 指向与

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

其配对的从设备的 tty_driver 结构。与此相似，在 tty_struct 结构中也有个指针 link，可以用来互相指向对方建立起伪终端主 / 从设备间的连接（见 910～911 行）。这样，在打开主设备的同时就分配好了从设备的 tty_struct 结构，下一次要打开从设备的时候就会在前面的 828 行发现所需的 tty_struct 结构已经存在，从而跳过创建 tty_struct 结构等操作，转到 fast_track 处（在 956 行）。

我们继续往下看（drivers/char/tty_io.c）：

==================== drivers/char/tty_io.c 914 1005 ====================

[tty_open()>init_dev()]

```
914      /*
915       * All structures have been allocated, so now we install them.
916       * Failures after this point use release_mem to clean up, so
917       * there's no need to null out the local pointers.
918       */
919      driver->table[idx] = tty;
920
921      if (!*tp_loc)
922           *tp_loc = tp;
923      if (!*ltp_loc)
924           *ltp_loc = ltp;
925      tty->termios = *tp_loc;
926      tty->termios_locked = *ltp_loc;
927      (*driver->refcount)++;
928      tty->count++;
929
930      /*
931       * Structures all installed ... call the ldisc open routines.
932       * If we fail here just call release_mem to clean up.  No need
933       * to decrement the use counts, as release_mem doesn't care.
934       */
935      if (tty->ldisc.open) {
936           retval = (tty->ldisc.open)(tty);
937           if (retval)
938                goto release_mem_out;
939      }
940      if (o_tty && o_tty->ldisc.open) {
941           retval = (o_tty->ldisc.open)(o_tty);
942           if (retval) {
943                if (tty->ldisc.close)
944                     (tty->ldisc.close)(tty);
945                goto release_mem_out;
946           }
947      }
948      goto success;
949
950      /*
951       * This fast open can be used if the tty is already open.
952       * No memory is allocated, and the only failures are from
953       * attempting to open a closing tty or attempting multiple
```

```
	 * opens on a pty master.
	 */
fast_track:
	if (test_bit(TTY_CLOSING, &tty->flags)) {
		retval = -EIO;
		goto end_init;
	}
	if (driver->type == TTY_DRIVER_TYPE_PTY &&
	    driver->subtype == PTY_TYPE_MASTER) {
		/*
		 * special case for PTY masters: only one open permitted,
		 * and the slave side open count is incremented as well.
		 */
		if (tty->count) {
			retval = -EIO;
			goto end_init;
		}
		tty->link->count++;
	}
	tty->count++;
	tty->driver = *driver; /* N.B. why do this every time?? */

success:
	*ret_tty = tty;

	/* All paths come through here to release the semaphore */
end_init:
	up_tty_sem(idx);
	return retval;

	/* Release locally allocated memory ... nothing placed in slots */
free_mem_out:
	if (o_tp)
		kfree(o_tp);
	if (o_tty)
		free_tty_struct(o_tty);
	if (ltp)
		kfree(ltp);
	if (tp)
		kfree(tp);
	free_tty_struct(tty);

fail_no_mem:
	retval = -ENOMEM;
	goto end_init;

	/* call the tty release_mem routine to clean out this slot */
release_mem_out:
	printk("init_dev: ldisc open failed, clearing slot %d\n", idx);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1003      release_mem(tty, idx);
1004      goto end_init;
1005 }
```

这里将新创建的 tty_struct 结构“安装”在其所属 tty_driver 结构的 table[]数组中相应的位置上（919 行），将新创建的 termios 结构安装在相应的数组中以及 tty_struct 结构中（921 行和 925 行）。最终完成对 tty_struct 结构的初始化以后，可能还要执行下层的打开文件操作。如果 tty_ldisc 数据结构中的函数指针 open 非 0，就要通过这个函数进行链路的初始化。如前所述，不管是哪一种终端设备，开始时总是采用与下标 N_TTY 对应的 tty_ldisc 结构。实际上，这个结构是 tty_ldisc_N_TTY，其中的指针 open 指向 n_tty_open()，这个函数的代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 860 877 ====================

[tty_open()>init_dev()>n_tty_open()]

```
860  static int n_tty_open(struct tty_struct *tty)
861  {
862       if (!tty)
863            return -EINVAL;
864
865       if (!tty->read_buf) {
866            tty->read_buf = alloc_buf();
867            if (!tty->read_buf)
868                 return -ENOMEM;
869       }
870       memset(tty->read_buf, 0, N_TTY_BUF_SIZE);
871       reset_buffer_flags(tty);
872       tty->column = 0;
873       n_tty_set_termios(tty, 0);
874       tty->minimum_to_wake = 1;
875       tty->closing = 0;
876       return 0;
877  }
```

读者在前面看到，tty_driver 结构中有个成分 flip，是个 tty_flip_buffer 数据结构，它就是终端设备的输入缓冲区。终端设备的输入有两种方式，一种叫“原始模式”，另一种叫“加工模式”，在实际应用中大多工作于加工模式。Unix 的作者给“加上模式”取了个很形象的名字，叫“ cooked”即“经过烹调的”。它把从终端设备键盘接收到的字符流比喻作”生米”，而把经过处理以后的字符流比喻成“熟饭”。如果把 tty_driver 结构中的 tty_flip_buffer 比喻作煮饭的锅，那么还得有个盘了来存放煮熟了的饭。为了这个目的，在 tty_driver 结构中设置了一个指针 read_buf，让它指向一个缓冲区，这就是盛饭的盘子。在打开终端设备时，如果发现这个缓冲区尚未分配就要为之分配一个（866 行），缓冲区的大小为一个页面。与这个缓冲页面相对应，在 tty_struct 结构中还有个位图 read_flags[]，位图中的每一位对应着上述缓冲区中的每个字符，在“规范模式”下用来分隔不同的缓冲行。此外，还有一些配合缓冲区使用的字段，例如 read_head 和 canon_read，都是缓冲区中当前读出位置的下标，等等。所有这些字段都要加以初始化，这是由 reset_buffer_flags()完成的（见 drivers/char/n_tty.c）：

==================== drivers/char/n_tty.c 118 133 ====================

[tty_open()>init_dev()>n_tty_open()>reset_buffer_flags()]

```
/*
 * Reset the read buffer counters, clear the flags,
 * and make sure the driver is unthrottled. Called
 * from n_tty_open() and n_tty_flush_buffer().
 */
static void reset_buffer_flags(struct tty_struct *tty)
{
     unsigned long flags;

     spin_lock_irqsave(&tty->read_lock, flags);
     tty->read_head = tty->read_tail = tty->read_cnt = 0;
     spin_unlock_irqrestore(&tty->read_lock, flags);
     tty->canon_head = tty->canon_data = tty->erasing = 0;
     memset(&tty->read_flags, 0, sizeof tty->read_flags);
     check_unthrottle(tty);
}
```

所涉及字段的作用在阅读 tty_read()的代码时就会清楚，现在只是初始化。

在 n_tty_open()中还有个函数要执行，那就是 n_tty_set_termios()。它的作用主要是根据终端设备的 termios 数据结构设置其 tty_struct 结构中的字符位图 process_char_map 和其它几个标志位（而不是“设置 termios 结构”）。位图 process_char_map 的大小是 32 字节，共 256 位，其中的每一位都对应着一个字符。如果位图中的某一位为 1，便说明与其对应的字符在“烹调”中需要加以特殊的处理。这个函数的代码也在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 786 849 ====================

[tty_opon()>init_dev()>n_tty_open()>n_tty_set_termios()]

```
static void n_tty_set_termios(struct tty_struct *tty, struct termios * old)
{
     if (!tty)
          return;

     tty->icanon = (L_ICANON(tty) != 0);
     if (test_bit(TTY_HW_COOK_IN, &tty->flags)) {
          tty->raw = 1;
          tty->real_raw = 1;
          return;
     }
     if (I_ISTRIP(tty) || I_IUCLC(tty) || I_IGNCR(tty) ||
         I_ICRNL(tty) || I_INLCR(tty) || L_ICANON(tty) ||
         I_IXON(tty) || L_ISIG(tty) || L_ECHO(tty) ||
         I_PARMRK(tty)) {
          cli();
          memset(tty->process_char_map, 0, 256/8);

          if (I_IGNCR(tty) || I_ICRNL(tty))
               set_bit('\r', &tty->process_char_map);
          if (I_INLCR(tty))
```

```
            set_bit('\n', &tty->process_char_map);

        if (L_ICANON(tty)) {
            set_bit(ERASE_CHAR(tty), &tty->process_char_map);
            set_bit(KILL_CHAR(tty), &tty->process_char_map);
            set_bit(EOF_CHAR(tty), &tty->process_char_map);
            set_bit('\n', &tty->process_char_map);
            set_bit(EOL_CHAR(tty), &tty->process_char_map);
            if (L_IEXTEN(tty)) {
                set_bit(WERASE_CHAR(tty),
                    &tty->process_char_map);
                set_bit(LNEXT_CHAR(tty),
                    &tty->process_char_map);
                set_bit(EOL2_CHAR(tty),
                    &tty->process_char_map);
                if (L_ECHO(tty))
                    set_bit(REPRINT_CHAR(tty),
                        &tty->process_char_map);
            }
        }
        if (I_IXON(tty)) {
            set_bit(START_CHAR(tty), &tty->process_char_map);
            set_bit(STOP_CHAR(tty), &tty->process_char_map);
        }
        if (L_ISIG(tty)) {
            set_bit(INTR_CHAR(tty), &tty->process_char_map);
            set_bit(QUIT_CHAR(tty), &tty->process_char_map);
            set_bit(SUSP_CHAR(tty), &tty->process_char_map);
        }
        clear_bit(__DISABLED_CHAR, &tty->process_char_map);
        sti();
        tty->raw = 0;
        tty->real_raw = 0;
    } else {
        tty->raw = 1;
        if ((I_IGNBRK(tty) || (!I_BRKINT(tty) && !I_PARMRK(tty))) &&
           (I_IGNPAR(tty) || !I_INPCK(tty)) &&
           (tty->driver.flags & TTY_DRIVER_REAL_RAW))
            tty->real_raw = 1;
        else
            tty->real_raw = 0;
    }
}
```

这个函数的代码中使用了许多宏定义，这些宏定义以后也常要用到。这些定义都在include/linux/tty.h和include/asm-i386/termbits.h中，虽然数量不少，却很有规则，先看几个基本操作：

==================== include/linux/tty.h 183 186 ====================

```
#define _I_FLAG(tty,f) ((tty)->termios->c_iflag & (f))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
184 #define _O_FLAG(tty,f) ((tty)->termios->c_oflag & (f))
185 #define _C_FLAG(tty,f) ((tty)->termios->c_cflag & (f))
186 #define _L_FLAG(tty,f) ((tty)->termios->c_lflag & (f))
```

这些宏操作分别检验 termios 结构中 c_iflag（输入）、c_oflag（输出）、c_cflag（控制）以及 c_lflag（本地）字段中的某一位。在这个基础上，例如，L_ICANON(tty)的定义为：

==================== include/linux/tty.h 230 230 ====================

```
230 #define L_ICANON(tty)  _L_FLAG((tty),ICANON)
```

就是说，前缀 L_表示要检查的标志位在 c_lflag 中，而标志位为 ICANON。由此类推，I_IGNCR 表示 c_iflag 中的 IGNCR 标志位，L_ECHO 表示 c_lflag 中的 ECHO 标志位，等等。这些标志位的意义和作用大致如下：

L_ICANON　表示“ canonical mode”或“规范模式”，这就是“加工模式”，对输入的字符要加以“烹调”。

L_ISTRIP　表示强制将输入字符的最高位设成 0，使它变成 7 位编码。

L_IUCLC　表示将接收到的大字字母转换成小写。

L_IGNCR　表示将接收到的“回车”字符“ \”丢弃。

L_ICRNL　表示将接收到的“ \r”字符转换成“ \n”（如果 I_IGNCR 为 0 的话）。

L_INLCR　表示将接收到的“ \n”字符转换成“ \r”。

L_IXON　表示如果接收到 CTRL_S 字符就暂停输出，直到接收到 CTRL_Q 时再恢复输出。

L_ISIG　表示如果接收到 CTRL_C 等控制字符就向在该终端设备上运行的进程发出信号。

L_ECHO　表示“本地回送”，就是在从终端设备接收到一个字符时就立即将该字符同送到该终端设备上。

此外，在不同的系统中或者不同的键盘上使用的控制字符有可能不同。例如，在屏幕上显示大量信息时，一般可以按 Ctrl-S 暂停屏幕上显示的信息向上滚动，然后可以按 Ctrl-Q 继续输出。可是，在有些键盘上也许不是 Ctrl-S 和 Ctrl-Q，而是别的什么。所以在 termios 结构中还有个数组 c_cc[]，用来定义各种控制字符的代码。下面是取自 include/linux/tty.h 和 include/asm-i386/termbits.h 的两个片段，可以用来说明这个数组的用途。

==================== include/linux/tty.h 173 174 ====================

```
173 #define START_CHAR(tty) ((tty)->termios->c_cc[VSTART])
174 #define STOP_CHAR(tty) ((tty)->termios->c_cc[VSTOP])
```

==================== include/asm-i386/termbits.h 29 30 ====================

```
29 #define VSTART 8
30 #define VSTOP 9
```

这表示在目标终端设备上用于这两种目的的控制字符分别是相应 termios 结构中的 c_cc[8]和 c_cc[9]。不过，这个数组不完全是用于控制字符，其中也有几个元素用于其它控制目的。读者在下一节中将会看到这些标志位和控制字符在“烹调”中的作用。

最后，init_dev()通过参数 ret_tty 返回指向 tty_struct 结构的指针。我们回到 tty_open()的代码中继续

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

往下看（drivers/char/tty_io.c）：

==================== drivers/char/tty_io.c 1361 1425 ====================

[tty_opon()]

```
1361 #ifdef CONFIG_UNIX98_PTYS
1362 init_dev_done:
1363 #endif
1364      filp->private_data = tty;
1365      file_move(filp, &tty->tty_files);
1366      check_tty_count(tty, "tty_open");
1367      if (tty->driver.type == TTY_DRIVER_TYPE_PTY &&
1368          tty->driver.subtype == PTY_TYPE_MASTER)
1369           noctty = 1;
1370 #ifdef TTY_DEBUG_HANGUP
1371      printk("opening %s...", tty_name(tty, buf));
1372 #endif
1373      if (tty->driver.open)
1374           retval = tty->driver.open(tty, filp);
1375      else
1376           retval = -ENODEV;
1377      filp->f_flags = saved_flags;
1378
1379      if (!retval && test_bit(TTY_EXCLUSIVE, &tty->flags) && !suser())
1380           retval = -EBUSY;
1381
1382      if (retval) {
1383 #ifdef TTY_DEBUG_HANGUP
1384           printk("error %d in opening %s...", retval,
1385                  tty_name(tty, buf));
1386 #endif
1387
1388           release_dev(filp);
1389           if (retval != -ERESTARTSYS)
1390                return retval;
1391           if (signal_pending(current))
1392                return retval;
1393           schedule();
1394           /*
1395            * Need to reset f_op in case a hangup happened.
1396            */
1397           filp->f_op = &tty_fops;
1398           goto retry_open;
1399      }
1400      if (!noctty &&
1401          current->leader &&
1402          !current->tty &&
1403          tty->session == 0) {
1404           task_lock(current);
1405           current->tty = tty;
```

```
1406            task_unlock(current);
1407            current->tty_old_pgrp = 0;
1408            tty->session = current->session;
1409            tty->pgrp = current->pgrp;
1410        }
1411        if ((tty->driver.type == TTY_DRIVER_TYPE_SERIAL) &&
1412            (tty->driver.subtype == SERIAL_TYPE_CALLOUT) &&
1413            (tty->count == 1)) {
1414            static int nr_warns;
1415            if (nr_warns < 5) {
1416                printk(KERN_WARNING "tty_io.c: "
1417                    "process %d (%s) used obsolete /dev/%s - "
1418                    "update software to use /dev/ttyS%d\n",
1419                    current->pid, current->comm,
1420                    tty_name(tty, buf), TTY_NUMBER(tty));
1421                nr_warns++;
1422            }
1423        }
1424        return 0;
1425 }
```

每个已打开文件都有个 file 结构作为代表，file 结构中有个 void 指针 private_data，对于常规的文件这个指针很少用到，现在就用来指向目标设备的 tty_struct 数据结构。这样，从当前进程到目标终端设备的连接就建立起来，。同时，tty_struct 结构中有个队列头 tty_files，这里通过 file_move()将指向该终端设备的 file 结构挂入这个队列，以便在需要时能找到所有与此相连的 file 结构。

前面已经通过相应 tty_ldisc 结构所提供的函数指针调用了与链路规则有关的 open 操作，可是具体的终端类型也可能有需要在打开文件时加以调用的函数（1373 行）。对于用作控制台的虚拟终端，其 tty_driver 数据结构为 console_driver，其 open 函数则为 con_open()，其代码在 drivers/char/console.c 中：

==================== drivers/char/console.c 2304 2328 ====================

[tty_open()>con_open()]

```
2304 /*
2305  * Allocate the console screen memory.
2306  */
2307 static int con_open(struct tty_struct *tty, struct file * filp)
2308 {
2309        unsigned int  currcons;
2310        int i;
2311
2312        currcons = MINOR(tty->device) - tty->driver.minor_start;
2313
2314        i = vc_allocate(currcons);
2315        if (i)
2316            return i;
2317
2318        vt_cons[currcons]->vc_num = currcons;
2319        tty->driver_data = vt_cons[currcons];
2320
```

```
    if (!tty->winsize.ws_row && !tty->winsize.ws_col) {
        tty->winsize.ws_row = video_num_lines;
        tty->winsize.ws_col = video_num_columns;
    }
    if (tty->count == 1)
        vcs_make_devfs (currcons, 0);
    return 0;
}
```

这个函数的作用是为当前的虚拟终端分配缓冲区和有关的数据结构，用来记录该虚拟终端的上下文，有兴趣的读者可自行阅读 vc_allocate()的代码（drivers/char/console.c）。此外，还要通过 vcs_make_devfs()在/dev 目录中创建一个 vcs 设备文件（节点），使应用软件在需要时可以绕过常规的终端设备界面，通过 vcs 设备文件直接访问这个缓冲区。

回到 tty_open()的代码中，余下的一些代码就留给读者了。

对于运行于“字符模式”的 VGA 卡，应用软件既可以通过常规的终端设备界面访问控制台，也可以通过 vcs 设备文件访问控制台。与此相似，如果 VGA 卡（SVGA 卡）运行于图像模式，则应用软件也可以把控制台作为图像设备打开，绕过常规的终端设备界面而直接访问图像缓冲区，称为“frame buffer”。这种安排有着特殊的意义，与前述的伪终端设备的使用相结合，就可以让视窗管理进程处理非拼音文字的显示。这样，就有了两种显示汉字的方法，一种是在内核中的虚拟终端（控制台）驱动程序中支持汉字的显示；另一种就是在应用软件中，即在视窗管理进程中支持汉字的显示。

由于篇幅的限制，我们不能在本书中讲到对作为图像设备的控制台，即图像缓冲区的驱动了，有兴趣或需要的读者可以自己阅读有关的代码，这些代码都在 drivers/video 目录下。

8.8 控制台的驱动

在本节中，我们通过 PC 机控制台的读操作来看虚拟终端，包括键盘和 VGA 显示卡的驱动。具体地说，就是从在键盘上输入个别的字符开始，到应用进程从其标准输入通道读取缓冲行为止的过程。虽然这只是个输入过程，但是实际上也包含了对屏幕的操作，因为终端设备的输入操作通常都包括了对输入字符的“回打”（echo）。也许可以说，在字符设备中（除一些网络设备外）控制台的驱动是最复杂的。我们之所以花比较大的篇幅来介绍控制台的驱动，一来正是因为它比较复杂，读者容易遇到问题；二来是因为控制台的驱动，从而对有关代码的理解，对于 Linux 的汉化工作是关键性的。

应用进程通常以其“控制终端”为“标准输入”通道，所以对“标准输入”通道的读操作就转化成对某个虚拟终端的读操作，最后落实成对控制台的读操作。经过读者已经熟知的过程，CPU 通过目标设备的 file_operations 数据结构进入控制台的读操作函数，这就是 tty_read()，定义于 drivers/char/tty_io.c：

==================== drivers/char/tty_io.c 645 667 ====================

```
static ssize_t tty_read(struct file * file, char * buf, size_t count,
                        loff_t *ppos)
{
    int i;
```

```
649       struct tty_struct * tty;
650       struct inode *inode;
651
652       /* Can't seek (pread) on ttys.  */
653       if (ppos != &file->f_pos)
654            return -ESPIPE;
655
656       tty = (struct tty_struct *)file->private_data;
657       inode = file->f_dentry->d_inode;
658       if (tty_paranoia_check(tty, inode->i_rdev, "tty_read"))
659            return -EIO;
660       if (!tty || (test_bit(TTY_IO_ERROR, &tty->flags)))
661            return -EIO;
662
663       /* This check not only needs to be done before reading, but also
664          whenever read_chan() gets woken up after sleeping, so I've
665          moved it to there.  This should only be done for the N_TTY
666          line discipline, anyway.  Same goes for write_chan(). -- jlc. */
667  #if 0
```

==================== drivers/char/tty_io.c 678 688 ====================

```
678  #endif
679       lock_kernel();
680       if (tty->ldisc.read)
681            i = (tty->ldisc.read)(tty,file,buf,count);
682       else
683            i = -EIO;
684       unlock_kernel();
685       if (i > 0)
686            inode->i_atime = CURRENT_TIME;
687       return i;
688  }
```

如前节中所述，在打开（设备）文件的过程中已经设置好了目标设备的 tty_struct 数据结构，并且使 file 结构中的指针 private_data 指向这个数据结构，所以这里 656 行毫不费力就找到了目标设备的 tty_struct 结构。

代表着终端设备的“链路规程”的是 tty_ldisc 数据结构；对于控制台，这就是序号为 N_TTY 的 tty_ldisc 结构 tty_ldisc_N_TTY，这是在初始化时通过 tty_register_ldisc()向系统登记的。681 行通过它所提供的函数指针 read 进入了驱动程序的下一个子层。这个数据结构定义于 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 1213 1230 ====================

```
1213 struct tty_ldisc tty_ldisc_N_TTY = {
1214      TTY_LDISC_MAGIC,   /* magic */
1215      "n_tty",       /* name */
1216      0,             /* num */
1217      0,             /* flags */
1218      n_tty_open,         /* open */
1219      n_tty_close,        /* close */
1220      n_tty_flush_buffer,     /* flush_buffer */
```

```
    n_tty_chars_in_buffer, /* chars_in_buffer */
    read_chan,         /* read */
    write_chan,        /* write */
    n_tty_ioctl,       /* ioctl */
    n_tty_set_termios, /* set_termios */
    normal_poll,       /* poll */
    n_tty_receive_buf, /* receive_buf */
    n_tty_receive_room,     /* receive_room */
    0             /* write_wakeup */
};
```

可见，其 read 函数为 read_chan()，其代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 925 962 ====================
[tty_read()>read_chan()]

```
static ssize_t read_chan(struct tty_struct *tty, struct file *file,
               unsigned char *buf, size_t nr)
{
    unsigned char *b = buf;
    DECLARE_WAITQUEUE(wait, current);
    int c;
    int minimum, time;
    ssize_t retval = 0;
    ssize_t size;
    long timeout;
    unsigned long flags;

do_it_again:

    if (!tty->read_buf) {
        printk("n_tty_read_chan: called with read_buf == NULL?!?\n");
        return -EIO;
    }

    /* Job control check -- must be done at start and after
      every sleep (POSIX.1 7.1.1.4). */
    /* NOTE: not yet done after every sleep pending a thorough
      check of the logic of this change. -- jlc */
    /* don't stop on /dev/console */
    if (file->f_dentry->d_inode->i_rdev != CONSOLE_DEV &&
        file->f_dentry->d_inode->i_rdev != SYSCONS_DEV &&
        current->tty == tty) {
        if (tty->pgrp <= 0)
            printk("read_chan: tty->pgrp <= 0!\n");
        else if (current->pgrp != tty->pgrp) {
            if (is_ignored(SIGTTIN) ||
               is_orphaned_pgrp(current->pgrp))
                return -EIO;
            kill_pg(current->pgrp, SIGTTIN, 1);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                return -ERESTARTSYS;
            }
        }

```

读者对上面这一段代码应该不会有困难了，我们把它留给读者结合前几章中的有关内容自己阅读。

==================== drivers/char/n_tty.c 963 994 ====================

[tty_read()>read_chan()]

```
963         minimum = time = 0;
964         timeout = MAX_SCHEDULE_TIMEOUT;
965         if (!tty->icanon) {
966               time = (HZ / 10) * TIME_CHAR(tty);
967               minimum = MIN_CHAR(tty);
968               if (minimum) {
969                     if (time)
970                           tty->minimum_to_wake = 1;
971                     else if (!waitqueue_active(&tty->read_wait) ||
972                            (tty->minimum_to_wake > minimum))
973                           tty->minimum_to_wake = minimum;
974               } else {
975                     timeout = 0;
976                     if (time) {
977                           timeout = time;
978                           time = 0;
979                     }
980                     tty->minimum_to_wake = minimum = 1;
981               }
982         }
983 
984         if (file->f_flags & O_NONBLOCK) {
985               if (down_trylock(&tty->atomic_read))
986                     return -EAGAIN;
987         }
988         else {
989               if (down_interruptible(&tty->atomic_read))
990                     return -ERESTARTSYS;
991         }
992 
993         add_wait_queue(&tty->read_wait, &wait);
994         set_bit(TTY_DONT_FLIP, &tty->flags);
```

在“原始”模式和非“规范”模式中，当输入缓冲区中有了最低限度的数据量 minimum_to_wake 时，就要唤醒正在等待着从该设备读出的进程。这里的 965～982 行为其确定一个合适的数值，这个数值一般都是 1。

对从同一终端设备的读出应该是互斥的，所以要放在临界区中。此外，还要在当前进程的系统堆栈中准备下一个 wait_queue_t 数据结构 wait，并把它挂入目标终端的等待队列 read_wait 中，使终端设备的驱动程序在有数据可读时可以唤醒这个进程。当然，也许终端设备的输入缓冲区中现在就有数据，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

因而根本就不需要进入睡眠，但是那也没有关系，读到了数据之后再把它从队列里摘除就可以了。这里还把 tty->flags 中的一个标志位 TTY_DONT_FLIP 设成 1，读者在后面将会看到这个标志位的意义和作用。

然后就是个 while 循环（drivers/char/n_tty.c）。

==================== drivers/char/n_tty.c 995 1037 ====================

[tty_read()>read_chan()]

```
      while (nr) {
            /* First test for status change. */
            if (tty->packet && tty->link->ctrl_status) {
                 unsigned char cs;
                 if (b != buf)
                      break;
                 cs = tty->link->ctrl_status;
                 tty->link->ctrl_status = 0;
                 put_user(cs, b++);
                 nr--;
                 break;
            }
            /* This statement must be first before checking for input
               so that any interrupt will set the state back to
               TASK_RUNNING. */
            set_current_state(TASK_INTERRUPTIBLE);

            if (((minimum - (b - buf)) < tty->minimum_to_wake) &&
                ((minimum - (b - buf)) >= 1))
                 tty->minimum_to_wake = (minimum - (b - buf));

            if (!input_available_p(tty, 0)) {
                 if (test_bit(TTY_OTHER_CLOSED, &tty->flags)) {
                      retval = -EIO;
                      break;
                 }
                 if (tty_hung_up_p(file))
                      break;
                 if (!timeout)
                      break;
                 if (file->f_flags & O_NONBLOCK) {
                      retval = -EAGAIN;
                      break;
                 }
                 if (signal_pending(current)) {
                      retval = -ERESTARTSYS;
                      break;
                 }
                 clear_bit(TTY_DONT_FLIP, &tty->flags);
                 timeout = schedule_timeout(timeout);
                 set_bit(TTY_DONT_FLIP, &tty->flags);
```

```
1036                continue;
1037            }
```

对于伪终端设备，可以通过系统调用 ioctl()将主 / 从两端的通信方式设置成“packet”（信包）模式，因为这两端往往在通过网络连接的不同计算机中。这这种情况下 tty->packet 为 1，而若 tty->link->ctrl_status 非 0 就表示有反映通信链路状态变化的控制信息需要递交，所以要优先读出这些控制信息（一些标志位）。不过这与我们这个情景无关。

指针 b 开始时（928 行）指向用户空间的缓冲区 buf，随着字符的读出而向前推进，所以(b-buf)就是已经读出的字符数。如果 tty->minimum_to_wake 开始时不是 1，那么在读出的过程中会向 1 逼近（1012~1014 行）。

接着通过 input_available_p()检查输入缓冲区中是否有数据（字符）可供读出。在“规范模式”下，检查的是经过加工以后的数量，而在原始模式下则是检查原始字符的数量。这个函数的代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 879 888 ====================

[tty_read()>read_chan()>input_available_p()]

```
879  static inline int input_available_p(struct tty_struct *tty, int amt)
880  {
881       if (tty->icanon) {
882            if (tty->canon_data)
883                 return 1;
884       } else if (tty->read_cnt >= (amt ? amt : 1))
885            return 1;
886
887       return 0;
888  }
```

如果缓冲区中还没有字符可供读出，则当前进程一般要睡眠等待。到缓冲区中有了可供读出的字符时才会被唤醒。在我们这个情景中，假定此时缓冲区中没有数据，所以当前进程进入了睡眠。

为了更好地把注意力集中在发生键盘中断以后的过程，我们暂且假定中断已经发生而缓冲区中已经有了数据，先来看当前进程被唤醒并被调度运行以后的操作（drivers/char/n_tty.c）。这样，等一下我们就可以集中看从键盘中断开始到唤醒睡眠中的进程为止的过程了。相比之下，那个过程更为重要，也更为复杂。

==================== drivers/char/n_tty.c 1038 1098 ====================

[tty_read()>read_chan()]

```
1038           current->state = TASK_RUNNING;
1039
1040           /* Deal with packet mode. */
1041           if (tty->packet && b == buf) {
1042                put_user(TIOCPKT_DATA, b++);
1043                nr--;
1044           }
1045
1046           if (tty->icanon) {
1047                /* N.B. avoid overrun if nr == 0 */
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
    while (nr && tty->read_cnt) {
        int eol;

        eol = test_and_clear_bit(tty->read_tail,
                &tty->read_flags);
        c = tty->read_buf[tty->read_tail];
        spin_lock_irqsave(&tty->read_lock, flags);
        tty->read_tail = ((tty->read_tail+1) &
                   (N_TTY_BUF_SIZE-1));
        tty->read_cnt--;
        spin_unlock_irqrestore(&tty->read_lock, flags);

        if (!eol || (c != __DISABLED_CHAR)) {
            put_user(c, b++);
            nr--;
        }
        if (eol) {
            /* this test should be redundant:
             * we shouldn't be reading data if
             * canon_data is 0
             */
            if (--tty->canon_data < 0)
                tty->canon_data = 0;
            break;
        }
    }
} else {
    int uncopied;
    uncopied = copy_from_read_buf(tty, &b, &nr);
    uncopied += copy_from_read_buf(tty, &b, &nr);
    if (uncopied) {
        retval = -EFAULT;
        break;
    }
}

/* If there is enough space in the read buffer now, let the
 * low-level driver know. We use n_tty_chars_in_buffer() to
 * check the buffer, as it now knows about canonical mode.
 * Otherwise, if the driver is throttled and the line is
 * longer than TTY_THRESHOLD_UNTHROTTLE in canonical mode,
 * we won't get any more characters.
 */
if (n_tty_chars_in_buffer(tty) <= TTY_THRESHOLD_UNTHROTTLE)
    check_unthrottle(tty);

if (b - buf >= minimum)
    break;
if (time)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1097                timeout = time;
1098        }
```

当前进程被唤醒时，一般在缓冲区中已经有了数据。我们在这里不关心 packet 模式的操作，所以跳过 1041～1044 行。先看规范模式（1046～1074 行）。在规范模式下，缓冲区中的字符是经过了加工的，要到累积起一个“缓冲行”，即碰到“\n”字符时才会唤醒等待读出的进程，此时 tty->read_cnt 表示缓冲行中的字符个数。另一方面，缓冲区 tty->read_buf[]是按环形缓冲区使用的（见 1055 行），tty->read_tail 指向当前可供读出的第一个字符。前一节中曾经讲到，tty_struct 结构中的 read_flags 是个位图，如果这个位图中对应于 tty->read_tail 这一位为 1，则表示这个位置上已经是缓冲行的终点，以后的数据已经属于另一缓冲行。所以，只要还没有到达终点，就逐个字符地调用 put_user()将其复制到用户空间去。这个过程要循环到已经满足了要求或者缓区中已经没有字符可读时才结束。这里的__DISABLED_CHAR 就是“\0”，缓冲行中的最后一个字符如果是__DISABLED_CHAR 就不复制到用户空间。这个字符定义于 include/linux/tty.h:

==================== include/linux/tty.h 125 130 ====================

```
/*
 * This character is the same as _POSIX_VDISABLE: it cannot be used as
 * a c_cc[] character, but indicates that a particular special character
 * isn't in use (eg VINTR has no character etc)
 */
#define __DISABLED_CHAR '\0'
```

再看非规范模式（1074～1082 行）。缓冲区同样也是 tty->read_buf[]，也按环形缓冲区使用，这些都一样，但是缓冲区中的字符是未经加工的，也无所谓“缓冲行”。另一方面，对于原始模式并没有不让把“\0”复制到用户空间的规定，所以这里通过 copy_from_read_buf()进行成片的复制，以加快速度，这个函数的代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 890 923 ====================
[tty_read()>read_chan()>copy_from_read_buf()]

```
/*
 * Helper function to speed up read_chan.  It is only called when
 * ICANON is off; it copies characters straight from the tty queue to
 * user space directly.  It can be profitably called twice; once to
 * drain the space from the tail pointer to the (physical) end of the
 * buffer, and once to drain the space from the (physical) beginning of
 * the buffer to head pointer.
 */
static inline int copy_from_read_buf(struct tty_struct *tty,
                          unsigned char **b,
                          size_t *nr)

{
     int retval;
     ssize_t n;
     unsigned long flags;

```

```
    retval = 0;
    spin_lock_irqsave(&tty->read_lock, flags);
    n = MIN(*nr, MIN(tty->read_cnt, N_TTY_BUF_SIZE - tty->read_tail));
    spin_unlock_irqrestore(&tty->read_lock, flags);
    if (n) {
        mb();
        retval = copy_to_user(*b, &tty->read_buf[tty->read_tail], n);
        n -= retval;
        spin_lock_irqsave(&tty->read_lock, flags);
        tty->read_tail = (tty->read_tail + n) & (N_TTY_BUF_SIZE-1);
        tty->read_cnt -= n;
        spin_unlock_irqrestore(&tty->read_lock, flags);
        *b += n;
        *nr -= n;
    }
    return retval;
}
```

由于缓冲区是环形的，缓冲着的字符有可能分成两段，所以要调用 copy_from_read_buf()两次。

缓冲区的大小总是有限的，如果从键盘打入字符的速度很快，而应用进程又来不及从缓冲区读出，则底层的驱动程序（主要是中断服务程序）可能已经因为缓冲区已满而暂时把“阀门”关闭了。现在，如果缓冲区中剩余的字符数量降到了“低水位”TTY_THRESHOLD_UNTHROTTLE 以下，则要通过 check_unthrottle()再打开阀门，这个函数的代码也在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 105 116 ====================

[tty_read()>read_chan()>check_unthrottle()]

```
/*
 * Check whether to call the driver.unthrottle function.
 * We test the TTY_THROTTLED bit first so that it always
 * indicates the current state.
 */
static void check_unthrottle(struct tty_struct * tty)
{
    if (tty->count &&
       test_and_clear_bit(TTY_THROTTLED, &tty->flags) &&
       tty->driver.unthrottle)
        tty->driver.unthrottle(tty);
}
```

除了清除代表着阀门的标志位 TTY_THROTTLED 以外，还可能要调用一个函数，具体取决于终端的 try_driver 数据结构。对于控制台终端的 tty_driver 数据结构 console_driver，这个函数是 con_unthrottle()，其代码在 drivers/char/console.c 中：

==================== drivers/char/n_tty.c 105 116 ====================

[tty_read()>read_chan()>check_unthrottle()>con_unthrottle()]

```
static void con_unthrottle(struct tty_struct *tty)
{
    struct vt_struct *vt = (struct vt_struct *) tty->driver_data;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

	wake_up_interruptible(&vt->paste_wait);
}
```

可见，目的只是唤醒可能在等待着要把数据写入缓冲区的进程。

回到 read_chan()的代码中继续往下看（drivers/char/n_tty.c）：

==================== drivers/char/n_tty.c 1099 1116 ====================

[tty_read()>read_chan()]

```
1099	clear_bit(TTY_DONT_FLIP, &tty->flags);
1100	up(&tty->atomic_read);
1101	remove_wait_queue(&tty->read_wait, &wait);
1102
1103	if (!waitqueue_active(&tty->read_wait))
1104		tty->minimum_to_wake = minimum;
1105
1106	current->state = TASK_RUNNING;
1107	size = b - buf;
1108	if (size) {
1109		retval = size;
1110		if (nr)
1111				clear_bit(TTY_PUSH, &tty->flags);
1112	} else if (test_and_clear_bit(TTY_PUSH, &tty->flags))
1113		 goto do_it_again;
1114
1115	return retval;
1116 }
```

当前进程既已读到了所要求的输入，需要放在临外区中进行的操作就完成了。这里的指针 buf 指向用户空间的缓冲区，而 b 指出向该缓冲区中的下一个空闲位置，所以 b-buf 就是已经读入该缓冲区中的字符数量。参数 nr 只是表明用户空间缓冲区的大小，即读出字符数量的上限，在规范模式下实际读出的字符数取决于具体的缓冲行。

标志位 TTY_PUSH 是由低层驱动程序在读到一个 EOF 字符并将其放入缓冲区时设置成 1 的，表示要让用户尽快把缓冲区的内容读走。如果此后从缓冲区读出了缓冲区中的所有字符，就把这个标志位清 0 并结束整个读出操作；否则，就说明还要继续从缓冲区读。所以如果本次读操作实际并未读出，则不让它结束，通过 1113 行的 goto 语句转到前面（937 行）的 do_it_again 处。如果本次读操作多少已经读出了若干字符则容许下次再读。

一般而言，一个典型的读终端过程可以分成下列三部分，或者这三部分的循环：

(1) 当前进程企图从终端的缓冲区读出，但是因为缓冲区中尚无足够字符供读出而受阻，进入睡眠；
(2) 然后，当使用者在键盘上输入字符，低层驱动程序将足够的字符写入缓冲区以后，就把睡眠中的进程唤醒；
(3) 睡眠中的进程被唤醒以后，继续完成读出。

我们在上面阅读的是这个过程中的第一、三两部分的代码，阅读时只是假定第二部分已经发生。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

实际上，第二部分才是设备驱动程序的主体，对于终端设备更是如此，下面我们就来看这一部分的代码。

使用者在键盘上按一个键，就产生了一个中断请求，CPU 在响应中断时便进入键盘的中断服务程序 keyboard_interrupt()。人们往往以为键盘是很简单的设备，但是实际上 PC 键盘的结构和操作都不简单。我们以字符“ A”为例来说明键盘的操作。当在键盘上按下一个键时，键盘立即就向母板发出一个字节的代码，称为“键盘扫描码”。具体的值取决于键的位置，“ A”健的键盘扫描码为 0x1c。母板上的键盘接口在接收到这个字节以后就把它转换成一种“系统扫描码”，并将其存入控制器的内部缓冲区，然后便向 CPU 发出一个中断请求。对于“ A”键，其系统扫描码为 0x1e。当 CPU 从键盘接口的数据寄存器读时，读出的就是系统扫描码。然后，当放开“ A”键时，键盘又要向母板发出键盘扫描码，而这一次的键盘扫描码是两个字节，第一个总是 0xf0，表示是放开一个键；然后是 0x1c，表示是哪一个键。同样，母板上的键盘接口也要把它转换成系统扫描码、也要向 CPU 发出中断请求，但是系统扫描码仍是一个字节，只是在 0x1e 的基础上把最高位设成 1，变成了 0x9e。这样，在 CPU 从键盘接口读出的单字节系统扫描码中，其最高位表示按下或放开，而低 7 位则与具体的键相对应。不光对普通的字符键是如此，对功能键和控制键也是一样。例如，左右两个 Shift 键的系统扫描码分别为 0x2a 和 0x36。这样，以输入一个小写的“ a”为例，CPU 实际上可能要发生 4 次中断、要依次从键盘接口读出 4 个字节的系统扫描码，例如：0x36，0x1e，0x9e，0xb6。至于把这个序列解释成什么，那就是软件的事了；如果解释成 ASCII 码，而键盘又没有锁定在大写状态，那就是“ a”。由此可见，键盘接口向 CPU 发出的中断请求并不意味着从键盘接收到了一个字符，而只是意味着键盘上发生了某种事件。此外，上面只是一般而言，实际上还有不少例外。其中最重要的是所谓“扩充键”。早期的 PC 键盘上只有 83 个键，后来扩充到了 101 键或 104 键，例如右边的 Ctrl 键就是一个扩充键。当按下或放开扩充键时，键盘扫描码和系统扫描码都以一个 0xe0 字节开头，所以按下右边 Ctrl 键时的键需扫描码是[0xe0，0x14]；放开时为[0xe0，0xf0，0x14]；相应的系统扫描码则为[0xe0，0x1d]以及[0xe0，0x9d]。

把这些事件和产生的代码列成表可以看得更清楚一些：

事件	键盘扫描码	系统扫描码
按下“ A”键	0x1c	0x1e
放开“ A”键	0xf0，0x1c	0x9e
按下右 Shift 键	0x59	0x36
放开右 Shift 键	0xf0，0x59	0xb6
按下右 Ctrl 键	0xe0，0x14	0xe0，0x1d
放开右 Ctrl 键	0xe0，0xf0，0x14	0xe0，0x9d

其它还有一些例外，主要与控制键和锁定键有关，我们在这里就不详细介绍。

之所以把键盘扫描码转换成系统扫描码，是为了建立起一个统一的扫描码界面。这种转换是由键盘接口完成的，所以不占用 CPU 的时间。不过，有需要时也可以通过键盘控制器关闭这种转换。显然，对于软件而言键盘扫描码是不可见的，所以我们在下面讲到“扫描码”时都是指系统扫描码。此外，按下一个键时产生的扫描码常常称为“ make code”，而放开一个键时产生的扫描码则常常称为“ break code”。

函数 keyboard_interrupt()的代码在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 479 488 ====================

```
static void keyboard_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
#ifdef CONFIG_VT
     kbd_pt_regs = regs;
#endif

     spin_lock_irq(&kbd_controller_lock);
     handle_kbd_event();
     spin_unlock_irq(&kbd_controller_lock);
}
```

为防止不同的 CPU 同时对键盘操作（多处理器系统中），内核中为键盘操作设立了一把锁，即 kbd_controller_lock，以保证键盘操作的互斥性。函数 handle_kbd_event()的代码在同一文件中：

==================== drivers/char/pc_keyb.c 446 476 ====================
[keyboard_interrupt()>handle_kbd_event()]

```
static unsigned char handle_kbd_event(void)
{
     unsigned char status = kbd_read_status();
     unsigned int work = 10000;

     while ((--work > 0) && (status & KBD_STAT_OBF)) {
          unsigned char scancode;

          scancode = kbd_read_input();

          /* Error bytes must be ignored to make the
             Synaptics touchpads compaq use work */
#if 1
          /* Ignore error bytes */
          if (!(status & (KBD_STAT_GTO | KBD_STAT_PERR)))
#endif
          {
               if (status & KBD_STAT_MOUSE_OBF)
                    handle_mouse_event(scancode);
               else
                    handle_keyboard_event(scancode);
          }

          status = kbd_read_status();
     }

     if (!work)
          printk(KERN_ERR "pc_keyb: controller jammed (0x%02X).\n", status);

     return status;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

就像其它设备一样，键盘接口上也有“控制 / 状态寄存器”和“数据寄存器”。首先要读出状态寄存器，看看发生了什么事。状态寄存器中有个标志位 KBD_STAT_OBF（OBF 表示“ Output Buffer Full”），当这个标志位为 1 时就表示键盘的内部缓冲区中有数据，可以通过数据寄存器读出。这里的 kbd_read_status()和 kbd_read_input()都是定义于 include/asm-i386/keyboard.h 中的宏操作：

==================== include/asm-i386/keyboard.h 48 49 ====================

```
#define kbd_read_input() inb(KBD_DATA_REG)
#define kbd_read_status() inb(KBD_STATUS_REG)
```

前面讲过，从 PC 键盘读入的是扫描码，代表着相应的键在键盘上的位置以及键的状态，而与具体文字的编码无关，因此要根据一定的规则将扫描码转换成相应的代码。同时，缓冲区中可能有不止一个字节，因而要通过一个 while 循环从缓冲区逐个字节地读出。鼠标器通常与键盘共用同一个接口，所以还要根据状态寄存器的内容确定数据的来源，我们在这里假定确实来自键盘。从键盘的数据寄存器读出了全部字符以后，状态寄存器中的标志位 KBD_STAT_OBF 就变成 0。如果读了 10000 次以后这个标志位还是 1，那当然是有问题了。对从键盘读入的每个字符都通过 handle_keyboard_event()进一步处理，其代码也在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 429 438 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()]

```
static inline void handle_keyboard_event(unsigned char scancode)
{
#ifdef CONFIG_VT
     kbd_exists = 1;
     if (do_acknowledge(scancode))
          handle_scancode(scancode, !(scancode & 0x80));
#endif
     tasklet_schedule(&keyboard_tasklet);
}

```

键盘并不是一种“只读”的设备，对键盘也有输出操作。键盘在收到数据后都要送回一个 KBD_REPLY_ACK（定义为 0xfa）予以确认，或送回一个 KBD_REPLY_RESEND（定义为 0xfe）要求重发，而 CPU 必须将其与正常的输入区分开来。为了这个目的，内核中设了一个全局量 reply_expected，每当发送一个字节给键盘时就将 reply_expected 设成 1，而在 handle_keyboard_event()中若发现 reply_expected 为 1 就要把输入丢弃。这就是这里调用 do_acknowledge()的目的，其代码在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 265 289 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>do_acknowledge()]

```
static int do_acknowledge(unsigned char scancode)
{
     if (reply_expected) {
       /* Unfortunately, we must recognise these codes only if we know they
        * are known to be valid (i.e., after sending a command), because there
        * are some brain-damaged keyboards (yes, FOCUS 9000 again) which have
        * keys with such codes :(
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
272         */
273         if (scancode == KBD_REPLY_ACK) {
274             acknowledge = 1;
275             reply_expected = 0;
276             return 0;
277         } else if (scancode == KBD_REPLY_RESEND) {
278             resend = 1;
279             reply_expected = 0;
280             return 0;
281         }
282         /* Should not happen... */
283 #if 0
284         printk(KERN_DEBUG "keyboard reply expected - got %02x\n",
285                scancode);
286 #endif
287     }
288     return 1;
289 }
```

对扫描码的实际处理是由 handle_scancode()完成的，其代码在 drivers/char/keyboard.c 中。这个函数比较长，我们分段阅读：

==================== drivers/char/keyboard.c 201 237 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()]

```
201 void handle_scancode(unsigned char scancode, int down)
202 {
203     unsigned char keycode;
204     char up_flag = down ? 0 : 0200;
205     char raw_mode;
206 
207     pm_access(pm_kbd);
208 
209     do_poke_blanked_console = 1;
210     tasklet_schedule(&console_tasklet);
211     add_keyboard_randomness(scancode | up_flag);
212 
213     tty = ttytab? ttytab[fg_console]: NULL;
214     if (tty && (!tty->driver_data)) {
215         /*
216          * We touch the tty structure via the the ttytab array
217          * without knowing whether or not tty is open, which
218          * is inherently dangerous.  We currently rely on that
219          * fact that console_open sets tty->driver_data when
220          * it opens it, and clears it when it closes it.
221          */
222         tty = NULL;
223     }
224     kbd = kbd_table + fg_console;
225     if ((raw_mode = (kbd->kbdmode == VC_RAW))) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            put_queue(scancode | up_flag);
            /* we do not return yet, because we want to maintain
               the key_down array, so that we have the correct
               values when finishing RAW mode or when changing VT's */
       }

       /*
        *  Convert scancode to keycode
        */
       if (!kbd_translate(scancode, &keycode, raw_mode))
           return;

```

参数 down 为 1 表示扫描码的最高位为 0，表示键处于按下状态。局部量 up_flag 实际上相当于把扫描码的最高位抽了出来。

这里的 pm_access()是为电源管理留下的一个口子，其意图是在人机界面上长时间没有活动以后就使显示器进入省电模式，然后一旦有键盘输入时就恢复到正常运行。不过，在目前的代码中这是一个空函数。

我们在第 3 章中讲过，中断服务程序不宜很长，有些可能比较费时的操作应该放在比较宽松的 bh 函数或 tasklet 中完成。对虚拟控制台的切换就是这样一种操作，所以控制台操作有一个 tasklet，那就是 console_tasklet()。这个 tasklet 是准备在执行完键盘中断服务程序以后，在从中断响应返回之前执行的，这里要通过 tasklet_schedule()，将其挂入相应的队列中。这里还调用了一个函数 add_keyboard_randomness()，这是借键盘输入的随机性加大系统中伪随机数的随机性，读者在前几章中也看到过类似的运用。

PC 机的控制台终端由显示器（图形卡）和键盘两部分构成，所以除 tty_struct 数据结构外还有个 kbd_struct 数据结构。同时，物理的显示器和键盘又可用于多个虚拟终端，通过 Alt 键与功能键的组合来切换。显然，每个虚拟终端都应该有自己的 tty_struct 结构和 kbd_struct 结构。为此，内核中设立了 ttytab[]和 kbd_table[]两个结构数组，而全局最 fg_console 则记录着当前的“前台”虚拟终端号。同时，为便于后面的处理，又设立了 tty 和 kbd 两个全局量，使它们分别指向“前台”虚拟终端的 tty_struct 结构和 kbd_struct 结构。

如果前台终端的键盘工作于“原始”模式 VC_RAW（可以通过系统调用 ioctl()设置），那就直接把扫描码放到键盘的接收队列中，否则就要将扫描码转换成“键码”后才放到队列中。所谓“原始”（raw）模式对于不同的层次有不同的意义，键盘的原始模式有两种，其中最原始的就是 VC_RAW，表示直接把扫描码送给应用层。

我们先看转换扫描码的过程 kbd_translate()，这个函数在 include/asm-i386/keyboard.h 中定义成 pckbd_translate()：

==================== include/asm-i386/keyboard.h 34 34 ====================

```
#define kbd_translate       pckbd_translate
```

这就是 PC 键盘的键码转换函数，其代码在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 291 359 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()>pckbd_translate()]

```
int pckbd_translate(unsigned char scancode, unsigned char *keycode,
                char raw_mode)
{
     static int prev_scancode;

     /* special prefix scancodes.. */
     if (scancode == 0xe0 || scancode == 0xe1) {
          prev_scancode = scancode;
          return 0;
     }

     /* 0xFF is sent by a few keyboards, ignore it. 0x00 is error */
     if (scancode == 0x00 || scancode == 0xff) {
          prev_scancode = 0;
          return 0;
     }

     scancode &= 0x7f;

     if (prev_scancode) {
       /*
        * usually it will be 0xe0, but a Pause key generates
        * e1 1d 45 e1 9d c5 when pressed, and nothing when released
        */
       if (prev_scancode != 0xe0) {
           if (prev_scancode == 0xe1 && scancode == 0x1d) {
             prev_scancode = 0x100;
             return 0;
           } else if (prev_scancode == 0x100 && scancode == 0x45) {
             *keycode = E1_PAUSE;
             prev_scancode = 0;
           } else {
#ifdef KBD_REPORT_UNKN
             if (!raw_mode)
               printk(KERN_INFO "keyboard: unknown e1 escape sequence\n");
#endif
             prev_scancode = 0;
             return 0;
           }
       } else {
           prev_scancode = 0;
           /*
            *  The keyboard maintains its own internal caps lock and
            *  num lock statuses. In caps lock mode E0 AA precedes make
            *  code and E0 2A follows break code. In num lock mode,
            *  E0 2A precedes make code and E0 AA follows break code.
            *  We do our own book-keeping, so we will just ignore these.
            */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
339            /*
340             *  For my keyboard there is no caps lock mode, but there are
341             *  both Shift-L and Shift-R modes. The former mode generates
342             *  E0 2A / E0 AA pairs, the latter E0 B6 / E0 36 pairs.
343             *  So, we should also ignore the latter. - aeb@cwi.nl
344             */
345            if (scancode == 0x2a || scancode == 0x36)
346            return 0;
347
348            if (e0_keys[scancode])
349            *keycode = e0_keys[scancode];
350            else {
351  #ifdef KBD_REPORT_UNKN
352              if (!raw_mode)
353                printk(KERN_INFO "keyboard: unknown scancode e0 %02x\n",
354                     scancode);
355  #endif
356              return 0;
357            }
358        }
359      } else if (scancode >= SC_LIM) {
```

首先，如前所述，如果读入的扫描码是 0xe0（或 0xe1），那就是扩充键的前缀码。此时的扫描码是个序列，所以需要为之实现一种“有限状态机”，全局量 prev_scancode 就是用于这个目的。这里先把前缀码作为一种状态保存在 prev_scancode 中，并返回 0，表示这个字节应予丢弃，而 handle_scancode() 也就随之返回（236 行）。此外，0x00 和 0xff 不是有效的扫描码，也要丢弃。

对于扫描码本身的处理因是否扩充键而异。如果 prev_scancode 非 0，那就说明在此之前的字节是个前缀码，此时又要看前缀码是否为 0xe0。

前缀码 0xe1 是个特例，在按下或放开 Pause 键的时候，键盘向主机发出一个三字节序列[0xe1，0x1d，0x45]或[0xe1，0x1d，0xc5]。所以，代码中为这个序列设置了个中间状态 0x100，316～329 行就是对这个特殊序列的检验。如果三个字节都对，那就是 E1_PAUSE，否则就予以丢弃。除 Pause 键以外，其它扩充键的前缀码都是 0xe0。前面讲到，左右两个 Shift 键并非扩充键，但是有些键盘在 NumLock 或 CapsLock 状态下操作左右 Shift 键时会把它们当成扩充键。由于 Linux 内核自己维持各种锁定状态，所以丢弃作为扩充键的左右 Shift 键扫描码（345～346 行）。

对于扩充键，从扫描码到键码的转换由定义于 drivers/char/pc_keyb.c 的数组 e0_keys[]提供：

==================== drivers/char/pc_keyb.c 227 244 ====================

```
static unsigned char e0_keys[128] = {
  0, 0, 0, 0, 0, 0, 0, 0,                         /* 0x00-0x07 */
  0, 0, 0, 0, 0, 0, 0, 0,                         /* 0x08-0x0f */
  0, 0, 0, 0, 0, 0, 0, 0,                         /* 0x10-0x17 */
  0, 0, 0, 0, E0_KPENTER, E0_RCTRL, 0, 0,         /* 0x18-0x1f */
  0, 0, 0, 0, 0, 0, 0, 0,                         /* 0x20-0x27 */
  0, 0, 0, 0, 0, 0, 0, 0,                         /* 0x28-0x2f */
  0, 0, 0, 0, 0, E0_KPSLASH, 0, E0_PRSCR,         /* 0x30-0x37 */
  E0_RALT, 0, 0, 0, 0, E0_F13, E0_F14, E0_HELP,   /* 0x38-0x3f */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
  E0_DO, E0_F17, 0, 0, 0, 0, E0_BREAK, E0_HOME,          /* 0x40-0x47 */
  E0_UP, E0_PGUP, 0, E0_LEFT, E0_OK, E0_RIGHT, E0_KPMINPLUS, E0_END,/* 0x48-0x4f */
  E0_DOWN, E0_PGDN, E0_INS, E0_DEL, 0, 0, 0, 0,          /* 0x50-0x57 */
  0, 0, 0, E0_MSLW, E0_MSRW, E0_MSTM, 0, 0,          /* 0x58-0x5f */
  0, 0, 0, 0, 0, 0, 0, 0,                  /* 0x60-0x67 */
  0, 0, 0, 0, 0, 0, 0, E0_MACRO,           /* 0x68-0x6f */
  0, 0, 0, 0, 0, 0, 0, 0,                  /* 0x70-0x77 */
  0, 0, 0, 0, 0, 0, 0, 0                   /* 0x78-0x7f */
};
```

例如，右 Ctrl 键的扫描码为[0xe0，0x1d]，所以用 0x1d 为下标从表中查得其键码 E0_RCTRL。又如 PageUp 键的扫描码为[0xe0，0x49]，所以用 0x49 为下标从表中查得其键码 E0_PGUP。这些键码也都定义于同一文件（drivers/char/pc_keyb.c）中：

==================== drivers/char/pc_keyb.c 131 158 ====================

```
/*
 * Translation of escaped scancodes to keycodes.
 * This is now user-settable.
 * The keycodes 1-88,96-111,119 are fairly standard, and
 * should probably not be changed - changing might confuse X.
 * X also interprets scancode 0x5d (KEY_Begin).
 *
 * For 1-88 keycode equals scancode.
 */

#define E0_KPENTER 96
#define E0_RCTRL   97
#define E0_KPSLASH 98
#define E0_PRSCR   99
#define E0_RALT    100
#define E0_BREAK   101  /* (control-pause) */
#define E0_HOME    102
#define E0_UP      103
#define E0_PGUP    104
#define E0_LEFT    105
#define E0_RIGHT   106
#define E0_END     107
#define E0_DOWN    108
#define E0_PGDN    109
#define E0_INS     110
#define E0_DEL     111

#define E1_PAUSE   119
```

数组 e0_keys[]的内容是可以通过系统调用 ioctl()设置的，但是这里的注释说改了以后可能会使 X Window 软件弄糊涂。这也部分地回答了读者可能会有的疑问，就是为什么需要“键码”。

再看不带前缀的扫描码，即非扩充键的扫描码。

==================== drivers/char/pc_keyb.c 359 384 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

[keyboard_interrupt()>handle_kbd_enevt()>handle_keyboard_event()>handle_scancode()>pckbd_translate()]

```
359      } else if (scancode >= SC_LIM) {
360          /* This happens with the FOCUS 9000 keyboard
361             Its keys PF1..PF12 are reported to generate
362             55 73 77 78 79 7a 7b 7c 74 7e 6d 6f
363             Moreover, unless repeated, they do not generate
364             key-down events, so we have to zero up_flag below */
365          /* Also, Japanese 86/106 keyboards are reported to
366             generate 0x73 and 0x7d for \ - and \ | respectively. */
367          /* Also, some Brazilian keyboard is reported to produce
368             0x73 and 0x7e for \ ? and KP-dot, respectively. */
369
370        *keycode = high_keys[scancode - SC_LIM];
371
372        if (!*keycode) {
373            if (!raw_mode) {
374  #ifdef KBD_REPORT_UNKN
375                printk(KERN_INFO "keyboard: unrecognized scancode (%02x)"
376                       " - ignored\n", scancode);
377  #endif
378            }
379            return 0;
380        }
381      } else
382        *keycode = scancode;
383      return 1;
384  }
```

常数 SC_LIM 的值为 89，即 0x59。数值小于 SC_LIM 的扫描码与键码相同而无需转换（382 行）；否则便由数组 high_keys[]提供转换，转换时以(scancode - SC_LIM)为下标。这个数组的定义也在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 194 201 ====================

```
static unsigned char high_keys[128 - SC_LIM] = {
  RGN1, RGN2, RGN3, RGN4, 0, 0, 0,                   /* 0x59-0x5f */
  0, 0, 0, 0, 0, 0, 0, 0,                           /* 0x60-0x67 */
  0, 0, 0, 0, 0, FOCUS_PF11, 0, FOCUS_PF12,          /* 0x68-0x6f */
  0, 0, 0, FOCUS_PF2, FOCUS_PF9, 0, 0, FOCUS_PF3,    /* 0x70-0x77 */
  FOCUS_PF4, FOCUS_PF5, FOCUS_PF6, FOCUS_PF7,        /* 0x78-0x7b */
  FOCUS_PF8, JAP_86, FOCUS_PF10, 0                   /* 0x7c-0x7f */
};
```

这个范围中的大多是功能键，其键码的数值在 89～127 范围中，如 FOCUS_PF2 就是 PF2 键的键码（FOCUS_PF1 的数值为 85，与扫描码相同，所以不在这个数组中）。现在可以比较完整地回答为什么需要将扫描码转换成键码的问题了。带前缀 0xe0 和不带前缀的扫描码占据着两块 128 字节（实际上只有 126 字节）的区间，但是实际上两个区间都是很“稀疏”的。就键盘的大小来说，现在还在 120 键以下，如果一键一码则只需要 120 字节以下，完全可以合并到同一个 126 字节的区间中去。可是，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

又不宜简单地把前缀丢弃了事，因为那样至少会造成功能键与普通字符键穿插在一起而带来不便。比较好的办法当然是统一编排一下、定义出一个标准的键码，使功能键的代码都集中在一起，例如上述的 96～111。这样，不管是 PC 键盘也好、Macintoshi 键盘也好，一旦转换到键码就都一样，与具体的键盘无关了。这里还要指出，键码仍然只是一种中性的代码，而与具体的语言文字无关。

回到 handle_scancode()的代码中继续往下看（drivers/char/keyboard.c）：

==================== drivers/char/keyboard.c 238 269 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()]

```
238      /*
239       * At this point the variable `keycode' contains the keycode.
240       * Note: the keycode must not be 0 (++Geert: on m68k 0 is valid).
241       * We keep track of the up/down status of the key, and
242       * return the keycode if in MEDIUMRAW mode.
243       */
244
245      if (up_flag) {
246           rep = 0;
247           if(!test_and_clear_bit(keycode, key_down))
248               up_flag = kbd_unexpected_up(keycode);
249      } else
250           rep = test_and_set_bit(keycode, key_down);
251
252  #ifdef CONFIG_MAGIC_SYSRQ        /* Handle the SysRq Hack */
253      if (keycode == SYSRQ_KEY) {
254           sysrq_pressed = !up_flag;
255           return;
256      } else if (sysrq_pressed) {
257           if (!up_flag) {
258                handle_sysrq(kbd_sysrq_xlate[keycode], kbd_pt_regs, kbd, tty);
259                return;
260           }
261      }
262  #endif
263
264      if (kbd->kbdmode == VC_MEDIUMRAW) {
265           /* soon keycodes will require more than one byte */
266           put_queue(keycode + up_flag);
267           raw_mode = 1; /* Most key classes will be ignored */
268      }
269
```

前面讲过，局部量 up_flag 的值表明相应的键处于按下（down）还是放开（up）状态。当然，键盘上绝大部分的键都处于放开状态，但是也可能同时有几个键处于按下状态，所以代码中使用一个全局的位图 key_down 来记录各个键的状态。如果读入的扫描码表明相应的键处于按下状态，那就把位图 key_down 中的对应位设成 1；进一步，如果这一位原来就已经是 1，那就说明使用者按下这个键不放，所以键盘开始了自动重复功能（250 行）。反之，如果读入的扫描码表明相应的键处于放开状态，那就

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

把位图中的对应位设成 0；如果原来就是 0，就说明至少是漏了一个扫描码，所以说是 kbd_unexpected_up()。

作为编译选择项，对 SysRq 键可以定义一些特殊的操作，不过我们在这里不感兴趣。

如果键盘的运行模式是 VC_MEDIUMRAW，或者说是“半生半熟”、“半原始”，那么至此已可以把转换后的键码放入接收队列了（266 行）。这样，如果键盘运行于两种原始模式之一，应用进程从键盘（控制台）读到的就是扫描码或键码。在前一节中讲到的伪终端加窗口管理进程的系统结构中，只有窗口管理进程才会直接从键盘读，所以可以在这个进程的软件中处理从扫描码或键码向具体语言文字（如汉字）编码的转换。当然，也可以不让键盘运行于原始模式，而在内核中完成这最后一步转换。读者还应注意，这里所说的原始模式是对最底层的键盘驱动而言，区别在于是否把扫描码转换成目标文字的代码（如 ASCII），这与前面在“终端”一层上的非规范模式不同，那种“原始”模式后面我们还要讲到。

我们再往下看对键盘输入的最后一步转换，这一次是要转换成 ASCII 码（或其它拼音文字的代码，下同）。在正常模式下运行的键盘要到把输入转换成 ASCII 码以后才放入接收队列，有的则根本不放入接收队列。

==================== drivers/char/keyboard.c 270 324 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()]

```
270         /*
271          * Small change in philosophy: earlier we defined repetition by
272          *    rep = keycode == prev_keycode;
273          *    prev_keycode = keycode;
274          * but now by the fact that the depressed key was down already.
275          * Does this ever make a difference? Yes.
276          */
277 
278         /*
279          *  Repeat a key only if the input buffers are empty or the
280          *  characters get echoed locally. This makes key repeat usable
281          *  with slow applications and under heavy loads.
282          */
283         if (!rep ||
284             (vc_kbd_mode(kbd,VC_REPEAT) && tty &&
285              (L_ECHO(tty) || (tty->driver.chars_in_buffer(tty) == 0)))) {
286              u_short keysym;
287              u_char type;
288 
289              /* the XOR below used to be an OR */
290              int shift_final = (shift_state | kbd->slockstate) ^
291                  kbd->lockstate;
292              ushort *key_map = key_maps[shift_final];
293 
294              if (key_map != NULL) {
295                   keysym = key_map[keycode];
296                   type = KTYP(keysym);
297 
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            if (type >= 0xf0) {
                type -= 0xf0;
                if (raw_mode && ! (TYPES_ALLOWED_IN_RAW_MODE & (1 << type)))
                  return;
                if (type == KT_LETTER) {
                  type = KT_LATIN;
                  if (vc_kbd_led(kbd, VC_CAPSLOCK)) {
                      key_map = key_maps[shift_final ^ (1<<KG_SHIFT)];
                      if (key_map)
                        keysym = key_map[keycode];
                  }
                }
                (*key_handler[type])(keysym & 0xff, up_flag);
                if (type != KT_SLOCK)
                  kbd->slockstate = 0;
            } else {
                /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
                if (!up_flag && !raw_mode)
                  to_utf8(keysym);
            }
        } else {
            /* maybe beep? */
            /* we have at least to update shift_state */
#if 1            /* how? two almost equivalent choices follow */
            compute_shiftstate();
            kbd->slockstate = 0; /* play it safe */
#else
    . . . . . . .
```

==================== drivers/char/keyboard.c 329 332 ====================

```
#endif
        }
    }
}
```

对于正常状态下产生的键盘输入，即 rep 为 0 时，这一段代码的执行是无条件的。而对于因按下键不放而自动重复产生的键盘输入，则是有条件的，条件是键盘运行于 VC_REPEAT 模式，即允许自动重复，并且终端运行于“echo”模式，或者输入缓冲区已经空了（以前的输入都已被读走）。如果不满足这些条件，键盘输入就被丢弃了。

从键码向目标码 keysym 的转换也是通过数组实现的，以键码的数值为下标，相应的元素就给出目标码。由于键码的数值不超过 127，所以数组的大小为 128。数组的类型为 16 位无符号短整数，其高 8 位表示键的类型，低 8 位则为具体的代码。Linux 内核的“母语”是英语，所以目标码基本上是 ASCII，不过也可以通过系统调用 ioctl()“下载”其它码表。不过，从键码向目标码的转换并不是一个数组就可完成的，因为按下同一个键在不同的情况下应产生出不同的代码。例如，单独按下“A”键时应产生“A”的代码 0x41，而若同时按下 Shift 键（更确切地说是在此之前按下 Shift 键，并且尚未放开）就应产生“a”的代码 0x61。除此之外，还得考虑 Ctrl 键等其它辅助键，所以实际上需要好几个这样的数组。那么，有几个辅助键影响着目标码的产生呢？一般而言有 4 个，即 Alt 键、Ctrl 键、Altgr 键（右边的 Alt

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

键）以及 Shift 键。这些辅助键还可以按一定的规则组合，一起来影响目标代码的产生。内核代码中有个全局量 shift_state，以位图的形式记录当前处于按下状态的辅助键，其高 28 位均为 0，低 4 位则用于 4 个辅助键，依次为 Alt、Ctrl、Altgr、Shift。这样，理论上一共可以有 16 个不同的码表，不过有些组合实际上并不使用，所以实际使用的共有 7 个码表。为此，drivers/char/defkeymap.c 定义了一个指针数组 key_maps[]，以 shift_state 为下标就可以找到当前适用的码表。

==================== drivers/char/defkeymap.c 141 146 ====================

```
ushort *key_maps[MAX_NR_KEYMAPS] = {
	plain_map, shift_map, altgr_map, 0,
	ctrl_map, shift_ctrl_map, 0, 0,
	alt_map, 0, 0, 0,
	ctrl_alt_map, 0
};
```

例如，当没有按下任何辅助键时 shift_state 为 0，所以适用的码表为 plain_map[]；同时按下 Shift 键时 shift_state 为 1，所以适用的码表为 shif_t_map[]；同时按下 Alt 键和 Ctrl 键（如 Alt-Ctrl-B）时 shift_state 为 12，所以适用的码表为 ctrl_alt_map[]。具体的码表都在 drivers/char/defkeymap.c 中，这个文件是由工具生成的，编译内核代码时会根据一个码表描述文件 drivers/char/defkeymap.map 自动生成。系统运行时也可以通过系统调用 ioctl()下载、替换这些码表。系统中有个工具/usr/bin/loadkeys，可以在运行时生成码表并且下载，达到动态地改变码表（从而改变目标语言）的目的。

此外，有些特殊的键盘上有所谓“ sticky ”辅助键，按一下就将键盘锁定在某种辅助键状态（kbd->slockstate 变成 1），就好像老是按下 Shift 键不放一样，再按一下就又回到正常状态。有的键盘上还可以将辅助键的作用反转（kbd->lockstate 为 1）。所以，上面的 290～291 行根据当前的 shift_state 和 kbd->slockstate 及 kbd->lockstate 计算出一个下标 shift_final，然后用这个下标在 key_maps[]中找到适用的码表（292 行）。接着，如果相应的码表存在，就可以进一步以具体的键码为下标从码表中读出目标代码（295 行）。

由于篇幅的关系，我们不能在这里列出所有 7 个码表，而只列出第一个码表 plain_map[]，让读者有个具体的印象：

==================== drivers/char/defkeymap.c 8 25 ====================

```
u_short plain_map[NR_KEYS] = {
	0xf200,	0xf01b,	0xf031,	0xf032,	0xf033,	0xf034,	0xf035,	0xf036,
	0xf037,	0xf038,	0xf039,	0xf030,	0xf02d,	0xf03d,	0xf07f,	0xf009,
	0xfb71,	0xfb77,	0xfb65,	0xfb72,	0xfb74,	0xfb79,	0xfb75,	0xfb69,
	0xfb6f,	0xfb70,	0xf05b,	0xf05d,	0xf201,	0xf702,	0xfb61,	0xfb73,
	0xfb64,	0xfb66,	0xfb67,	0xfb68,	0xfb6a,	0xfb6b,	0xfb6c,	0xf03b,
	0xf027,	0xf060,	0xf700,	0xf05c,	0xfb7a,	0xfb78,	0xfb63,	0xfb76,
	0xfb62,	0xfb6e,	0xfb6d,	0xf02c,	0xf02e,	0xf02f,	0xf700,	0xf30c,
	0xf703,	0xf020,	0xf207,	0xf100,	0xf101,	0xf102,	0xf103,	0xf104,
	0xf105,	0xf106,	0xf107,	0xf108,	0xf109,	0xf208,	0xf209,	0xf307,
	0xf308,	0xf309,	0xf30b,	0xf304,	0xf305,	0xf306,	0xf30a,	0xf301,
	0xf302,	0xf303,	0xf300,	0xf310,	0xf206,	0xf200,	0xf03c,	0xf10a,
	0xf10b,	0xf200,	0xf200,	0xf200,	0xf200,	0xf200,	0xf200,	0xf200,
	0xf30e,	0xf702,	0xf30d,	0xf01c,	0xf701,	0xf205,	0xf114,	0xf603,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        0xf118,  0xf601,  0xf602,  0xf117,  0xf600,  0xf119,  0xf115,  0xf116,
        0xf11a,  0xf10c,  0xf10d,  0xf11b,  0xf11c,  0xf110,  0xf311,  0xf11d,
        0xf200,  0xf200,  0xf200,  0xf200,  0xf200,  0xf200,  0xf200,  0xf200,
};
```

例如，“A”键的扫描码为 0x1e，因而键码也是 0x1e（因为小于 89），以此为下标从这个数组中读得其代码为 0xfb61。代码的高 8 位代表类型，低 8 位即为字符“a”的代码 0x61。当高 8 位的数值大于等于 0xf0 时，其数值与 0xf0 之差表示键的类型，小于 0xf0 时则表示采用 Unicode 的 UTF-8 编码。

我们先看键的类型，include/linux/keyboard.h 中共定义了 13 种类型：

==================== include/linux/keyboard.h 35 47 ====================

```
#define KT_LATIN   0    /* we depend on this being zero */
#define KT_LETTER  11   /* symbol that can be acted upon by CapsLock */
#define KT_FN      1
#define KT_SPEC         2
#define KT_PAD          3
#define KT_DEAD         4
#define KT_CONS         5
#define KT_CUR          6
#define KT_SHIFT   7
#define KT_META         8
#define KT_ASCII   9
#define KT_LOCK         10
#define KT_SLOCK   12
```

这里 KT_LATIN 表示普通可打印字符，KT_LETTER 表示字母，KT_FN 为功能键，KT_SPEC 为特殊键，KT_PAD 为“副键盘”或“数字键盘”上的键，KT_SHIFT 表示辅助键，等等。注意这里 KT_ASCII 并不表示“ASCII 码”，而是表示用来输入十六进制数字。还有，KT_DEAD 现在已经不用了，只是为兼容而还保留着。上面“A”键的类型码为 0xfb，所以是 KT_LETTER。

同到 handle_scancode()的代码中，宏操作 KTYP()从目标码中取出其高 8 位类型码。如果大于等于 0xf0 即为普通字符（299～312 行）。对于原始模式，输入的扫描码或键码已放入接收队列，但是对某些键的按下或放开也还需要一些附加操作，例如 Shift 键的状态就需要反映在 shift_state 中。这里的常数 TYPES_ALLOWED_IN_RAW_MODE 定义于 drivers/char/keyboard.c：

==================== drivers/char/keyboard.c 121 123 ====================

```
/* Key types processed even in raw modes */

#define TYPES_ALLOWED_IN_RAW_MODE ((1 << KT_SPEC) | (1 << KT_SHIFT))
```

这表示：即使是在原始模式下，对 KT_SPEC 和 KT_SHIFT 这两类键也需要进行一些处理，而其它的就不需要了。

至于正常的键盘操作，那就全都要经过进一步的处理了。其中有些输入要在经过处理以后放入接收队列，有的（如 Shift 键）则处理以后将其丢弃。首先，字母类（KT_LETTER）的输入与普通可打印字符（KT_LATIN）的区别仅在于 CapsLock 键的作用，所以这里把 KT_LETTER 替换成 KT_LATIN，并且根据当前键盘上的 CapsLock 发光二极管（Led）是否亮着决定是否将 Shift 键的作用反相。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

然后，就要根据键的类型调用相应的处理程序了。这里的 key_handler[]是个以类型为下标的函数指针数组，定义于 drivers/char/keyboard.c：

==================== drivers/char/keyboard.c 115 119 ====================

```
static k_hand key_handler[16] = {
        do_self, do_fn, do_spec, do_pad, do_dead, do_cons, do_cur, do_shift,
        do_meta, do_ascii, do_lock, do_lowercase, do_slock, do_dead2,
        do_ignore, do_ignore
};
```

数组的大小是 16，但是前面实际上只定义了 13 种类型，所以最后 3 个函数指针是不可能用到的。我们先看看对 KT_LATIN，即普通可打印字符的处理，函数 do_self()的代码在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 542 557 ====================

```
static void do_self(unsigned char value, char up_flag)
{
        if (up_flag)
                return;         /* no action, if this is a key release */

        if (diacr)
                value = handle_diacr(value);

        if (dead_key_next) {
                dead_key_next = 0;
                diacr = value;
                return;
        }

        put_queue(value);
}
```

首先，up_flag 非 0 表示当前的事件是放开（而不是按下）一个键，所以什么事也不用干，把输入丢弃就是了。否则，如果全局量 diacr 和 dead_key_next 都是 0，那就是一个正常的输入字符了，所以通过 put_queue()把这个字符放在接收队列中。

那么，diacr 和 dead_key_next 是干什么用的呢？原来，在许多拼音文字中都有所谓“accent”字符（也称为“diacritical”字符），例如“A”顶上加个小圈就是一个。输入这个特殊字符时先要同时按一下 Ctrl 和“.”键，然后按“o”键，最后按“A”键。由于这是个序列，所以又得要实现一个“有限状态机”，而 diacr 和 dead_key_next 就是用于这个目的。

以刚才讲的序列为例，“.”键的键码为 0x34，但是因为同时按 Ctrl 键，所以从 ctrl_map[]中（这里没有列出）读出目标码为 0xf20e，其类型为 KT_SPEC。于是，从 key_handler[]中找到函数指针为 do_spec()，其代码也在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 525 535 ====================

```
static void do_spec(unsigned char value, char up_flag)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (up_flag)
        return;
    if (value >= SIZE(spec_fn_table))
        return;
    if ((kbd->kbdmode == VC_RAW || kbd->kbdmode == VC_MEDIUMRAW) &&
        !(SPECIALS_ALLOWED_IN_RAW_MODE & (1 << value)))
        return;
    spec_fn_table[value]();
}
```

长话短说，这里最后又通过同一文件（drivers/char/keyboard.c）中的函数指针数组 spec_fn_table[] 执行一个函数：

==================== drivers/char/keyboard.c 132 138 ====================

```
static void_fnp spec_fn_table[] = {
    do_null,  enter,        show_ptregs,  show_mem,
    show_state,   send_intr,    lastcons, caps_toggle,
    num,      hold,         scroll_forw,  scroll_back,
    boot_it, caps_on, compose, SAK,
    decr_console, incr_console, spawn_console,     bare_num
};
```

由于目标码中的低 8 位为 0x0e，所以执行的是 compose()，还是在同一文件中：

==================== drivers/char/keyboard.c 488 491 ====================

```
static void compose(void)
{
    dead_key_next = 1;
}
```

这样，当输入"o"键时，在 do_self()中就因 dead_key_next 为 1 而将该字符的代码存放在 diacr 中，dead_key_next 则又清成 0。接着，当输入"A"键时则因 diacr 非 0 而调用 handle_diacr()，其代码仍在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 589 613 ====================

```
/*
 * We have a combining character DIACR here, followed by the character CH.
 * If the combination occurs in the table, return the corresponding value.
 * Otherwise, if CH is a space or equals DIACR, return DIACR.
 * Otherwise, conclude that DIACR was not combining after all,
 * queue it and return CH.
 */
unsigned char handle_diacr(unsigned char ch)
{
    int d = diacr;
    int i;

    diacr = 0;

```

```
603       for (i = 0; i < accent_table_size; i++) {
604           if (accent_table[i].diacr == d && accent_table[i].base == ch)
605               return accent_table[i].result;
606       }
607
608       if (ch == ' ' || ch == d)
609           return d;
610
611       put_queue(d);
612       return ch;
613   }
```

每个这样的合成字符都有一个数据结构，根据前后两个字符给出合成字符的代码，这些数据结构都在一个数组 accent_table[]中，所以这里搜索这个数组并返回结果（605 行）。返回的合成字符则在do_self()中放入接收队列。

我们在这里当然不可能逐一去读 key_handler[]中那些函数的代码，读者可以自己阅读。

回到 handle_scancode()的代码中，我们已经看了类型码大于等于 0xf0 时的处理。前面我们也提到过，如果类型码小于 0xf0 就表示采用 Unicode 的 UTF-8 编码。为方便阅读，我们再把 handle_scancode()中有关的几行列出于下：

==================== drivers/char/keyboard.c 313 317 ====================

```
313                   } else {
314                       /* maybe only if (kbd->kbdmode == VC_UNICODE) ? */
315                       if (!up_flag && !raw_mode)
316                         to_utf8(keysym);
317                   }
```

这里执行 to_utf8()的条件是不言自明的，to_utf8()的代码在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 171 184 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()>to_utf8()]

```
171   void to_utf8(ushort c) {
172       if (c < 0x80)
173         put_queue(c);           /*  0*******  */
174       else if (c < 0x800) {
175         put_queue(0xc0 | (c >> 6)); /*  110***** 10******  */
176         put_queue(0x80 | (c & 0x3f));
177       } else {
178         put_queue(0xe0 | (c >> 12));     /*  1110**** 10****** 10******  */
179         put_queue(0x80 | ((c >> 6) & 0x3f));
180         put_queue(0x80 | (c & 0x3f));
181       }
182       /* UTF-8 is defined for words of up to 31 bits,
183          but we need only 16 bits here */
184   }
```

这段代码正好可以让读者复习和消化前一节中讲到的 UTF-8 编码。注意这里没有对各种键作分类处理，而只是把转换成的代码依次放入接收队列，在这方面有点像原始模式。此外，这里的 Unicode

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

仍是一键一码，所以这些程序只能用于拼音文字。

再回到 handle_scancode()的代码中，最后还有一种可能，就是 key_maps[]中不存在由当前 shift_final 决定的码表（318～330 行）。既然没有码表，那就只好把输入丢弃。不过，前面已经根据最新的输入改变了当前的位图 key_down[]，也许最新的输入恰好是辅助键的状态变化，因而也应该反映在位图 shift_state 中，所以要根据位图 key_down[]的内容计算出最新的 shift_state。我们再列出 handle_scancode()中有关的几行如下：

==================== drivers/char/keyboard.c 322 323 ====================

```
322             compute_shiftstate();
323             kbd->slockstate = 0; /* play it safe */
```

这里 compute_shiftstate()的代码在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 737 763 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_keyboard_event()>handle_scancode()>compute_shiftstate()]

```
/* called after returning from RAW mode or when changing consoles -
   recompute k_down[] and shift_state from key_down[] */
/* maybe called when keymap is undefined, so that shiftkey release is seen */
void compute_shiftstate(void)
{
     int i, j, k, sym, val;

     shift_state = 0;
     for(i=0; i < SIZE(k_down); i++)
       k_down[i] = 0;

     for(i=0; i < SIZE(key_down); i++)
       if(key_down[i]) {     /* skip this word if not a single bit on */
         k = i*BITS_PER_LONG;
         for(j=0; j<BITS_PER_LONG; j++,k++)
           if(test_bit(k, key_down)) {
          sym = U(plain_map[k]);
          if(KTYP(sym) == KT_SHIFT || KTYP(sym) == KT_SLOCK) {
            val = KVAL(sym);
            if (val == KVAL(K_CAPSSHIFT))
              val = KVAL(K_SHIFT);
            k_down[val]++;
            shift_state |= (1<<val);
          }
           }
       }
}
```

这里依次扫描位图 key_down[]中的每一位，对当前处于按下状态的每一个键都从 plain_map[]中读出其代码，看看是什么类型。如果是辅助键或锁定键则将 shiftstate 中的相应位设成 1。

完成了 handle_scancode()的执行以后，对键盘中断的服务也就基本上完成了。

总而言之，键盘中断是由键盘上的事件引起的，根据键盘运行模式的不同，有些事件导致字符的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

接收，有的则只引起键盘驱动程序底层的某些状态变化。对于导致字符接收的事件，所接收的代码视键盘运行模式的不同和适用码表的不同而异，分别可以是扫描码、键码或者目标码。这些代码通常只是一个字节，但是在有些情况下也可能是个字节序列。然而，有一点是共同的，凡是接收的代码都通过 put_queue()逐个字节地放入“前台”键盘的接收队列。这个函数的代码在 drivers/char/keyboard.c 中：

==================== drivers/char/keyboard.c 335 342 ====================

```
void put_queue(int ch)
{
    wake_up(&keypress_wait);
    if (tty) {
        tty_insert_flip_char(tty, ch, 0);
        con_schedule_flip(tty);
    }
}
```

对于从键盘接收到而未经进一步加工的字符，先放在 tty_struct 结构内的一个所谓“ flip 缓冲区 ”中（include/linux/tty_flip.h）：

==================== include/linux/tty_flip.h 10 18 ====================

[put_queue()>tty_insert_flip_char()]

```
_INLINE_ void tty_insert_flip_char(struct tty_struct *tty,
                    unsigned char ch, char flag)
{
    if (tty->flip.count < TTY_FLIPBUF_SIZE) {
        tty->flip.count++;
        *tty->flip.flag_buf_ptr++ = flag;
        *tty->flip.char_buf_ptr++ = ch;
    }
```

在中断服务的过程中，CPU 对从键盘读入的数据进行了一些处理和转换。相对而言，这些处理和转换都是很简单的，可以在很短的时间内完成。可是，对于产生输入字符的事件来说，已经进行的这些处理还只是“万里长征走完了第一步”，接下去还要进行不少处理，而且那些处理就比较费时间了。我们在第 3 章中讲到过，对于这种本应在中断服务程序中进行，可是又比较费时间的操作应该放在 bh 函数中或者作为 tasklet 在比较宽松（允许中断）的条件下执行。所以，将字符放入 flip 缓冲区以后，还要通过 con_schedule_flip()调度控制台终端的 tasklet 运行（include/linux/kbd_kern.h）。控制台终端的 tasklet 是 console_tasklet()。所谓调度其运行，就是将一个指向这个函数的指针通过一个结构挂入 tasklet 的执行队列。

==================== include/linux/kbd_kern.h 164 168 ====================

[put_queue()>con_schedule_flip()]

```
extern inline void con_schedule_flip(struct tty_struct *t)
{
    queue_task(&t->flip.tqueue, &con_task_queue);
    tasklet_schedule(&console_tasklet);
}
```

这个 tasklet 定义于 drivers/char/console.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/char/console.c 2372 2372 ====================

```
2372 DECLARE_TASKLET_DISABLED(console_tasklet, console_softint, 0);
```

读者应结合第 3 章中的有关内容搞清其机理。这个 tasklet 的执行程是 console_softint()，其代码也在 drivers/char/console.c 中：

==================== drivers/char/console.c 2003 2045 ====================

```
2003 /*
2004  * This is the console switching tasklet.
2005  *
2006  * Doing console switching in a tasklet allows
2007  * us to do the switches asynchronously (needed when we want
2008  * to switch due to a keyboard interrupt).  Synchronization
2009  * with other console code and prevention of re-entrancy is
2010  * ensured with console_lock.
2011  */
2012 static void console_softint(unsigned long ignored)
2013 {
2014      /* Runs the task queue outside of the console lock.  These
2015       * callbacks can come back into the console code and thus
2016       * will perform their own locking.
2017       */
2018      run_task_queue(&con_task_queue);
2019
2020      spin_lock_irq(&console_lock);
2021
2022      if (want_console >= 0) {
2023           if (want_console != fg_console && vc_cons_allocated(want_console)) {
2024                hide_cursor(fg_console);
2025                change_console(want_console);
2026                /* we only changed when the console had already
2027                   been allocated - a new console is not created
2028                   in an interrupt routine */
2029           }
2030           want_console = -1;
2031      }
2032      if (do_poke_blanked_console) { /* do not unblank for a LED change */
2033           do_poke_blanked_console = 0;
2034           poke_blanked_console();
2035      }
2036      if (scrollback_delta) {
2037           int currcons = fg_console;
2038           clear_selection();
2039           if (vcmode == KD_TEXT)
2040                sw->con_scrolldelta(vc_cons[currcons].d, scrollback_delta);
2041           scrollback_delta = 0;
2042      }
2043
2044      spin_unlock_irq(&console_lock);
```

```
}
```

一进入 console_softint()，我们就上了一个层次，从键盘上升到了终端设备的层次。

使用者通过按 Alt 键和某个功能键（例如 F2）切换虚拟终端时，具体的切换也是在这里完成的，不过我们现在对此不感必趣。我们在这里关心的是对 run_task_queue()的调用。这个队列中有些什么函数要执行呢？前面我们看到，con_schedule_flip()中把相应 tty_struct 结构中的 flip.tqueue 挂入了这个队列，而在前一节中打开文件时调用的 initialize_tty_struct()里面，则将其函数指针设置成指向 flush_to_ldisc()，这个函数的代码在 drivers/char/tty_io.c 中：

==================== drivers/char/tty_io.c 1863 1901 ====================

[console_softint()>run_task_queue()>flush_to_ldisc()]

```
/*
 * This routine is called out of the software interrupt to flush data
 * from the flip buffer to the line discipline.
 */
static void flush_to_ldisc(void *private_)
{
	struct tty_struct *tty = (struct tty_struct *) private_;
	unsigned char *cp;
	char		*fp;
	int		count;
	unsigned long flags;

	if (test_bit(TTY_DONT_FLIP, &tty->flags)) {
		queue_task(&tty->flip.tqueue, &tq_timer);
		return;
	}
	if (tty->flip.buf_num) {
		cp = tty->flip.char_buf + TTY_FLIPBUF_SIZE;
		fp = tty->flip.flag_buf + TTY_FLIPBUF_SIZE;
		tty->flip.buf_num = 0;

		save_flags(flags); cli();
		tty->flip.char_buf_ptr = tty->flip.char_buf;
		tty->flip.flag_buf_ptr = tty->flip.flag_buf;
	} else {
		cp = tty->flip.char_buf;
		fp = tty->flip.flag_buf;
		tty->flip.buf_num = 1;

		save_flags(flags); cli();
		tty->flip.char_buf_ptr = tty->flip.char_buf + TTY_FLIPBUF_SIZE;
		tty->flip.flag_buf_ptr = tty->flip.flag_buf + TTY_FLIPBUF_SIZE;
	}
	count = tty->flip.count;
	tty->flip.count = 0;
	restore_flags(flags);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        tty->ldisc.receive_buf(tty, cp, fp, count);
}
```

这个函数的任务是通过 tty->ldisc.receive_buf()从 flip 缓冲区中把数据搬运到另一个缓冲区中，同时加以处理。

可是，这里隐含着一个问题。我们知道，tasklet 在执行过程中是允许中断的（否则就没有意义了），如果 flush_to_ldisc()正在从一个缓冲区往外读数据，而正好又发生了键盘中断，从而又通过 put_queue()往同一缓冲区里面写数据，那岂不是乱了套？可是，我们所熟知的一些保证互斥的办法在这里却都用不上。首先，tasklet 和中断服务程序都不是在某个进程的上下文中，因而不能睡眠。其次。加锁也行不通，因为这二者可能（在单 CPU 的系统中则肯定）由同一个 CPU 执行。当然，可以在执行的过程中关闭中断以免打扰，可是那样又违背了设置和使用 tasklet 机制的初衷。这里的解决办法是采用“双缓冲”，即交替使用两个缓冲区。这就好像翻来覆去使用一张纸的两面一样，我在用这一面就让你用另一面，等我用完这一面就再翻个面。正因为这样，才称之为“ flip”缓冲区，其数据结构定义于 include/linux/tty.h：

==================== include/linux/tty.h 139 149 ====================

```
struct tty_flip_buffer {
        struct tq_struct tqueue;
        struct semaphore pty_sem;
        char      *char_buf_ptr;
        unsigned char *flag_buf_ptr;
        int       count;
        int       buf_num;
        unsigned char char_buf[2*TTY_FLIPBUF_SIZE];
        char      flag_buf[2*TTY_FLIPBUF_SIZE];
        unsigned char slop[4]; /* N.B. bug overwrites buffer by 1 */
};
```

可见，缓冲区 char_buf[]的大小是 TTY_FLIPBUF_SIZE 的两倍，这就好像一张纸的两面。与 char_buf[]相平行还有一个数组 flag_buf[]，一个字符总是与一个 flag 字节配对的，不过从 put_queue()调用 tty_insert_flip_char()时总是把 flag 字节设成 0。指针 char_buf_ptr 和 flag_buf_ptr 都是给 tty_insert_flip_char()使用的，而 flush_to_ldisc()中对这些指针的设置就起着把纸翻一个面的作用。当然，这种方法也是有缺点的，那就是实际上把缓冲区的容量减小了一半；而 tty_insert_flip_char()在发现缓冲区已满的情况下就会把按收到的字符丢弃。所以，在 flush_to_ldisc()中保留了一种选择，如果把 tty_struct 结构中的标志位 TTY_DONT_FLIP 设成 1，那就把要执行的函数转到 tq_timer 队列中去，让它在时钟中断时在关中断的条件下执行。在前面 read_chan()的代码中，我们看到当高层从缓冲区读出时就把 TTY_DONT_FLIP 标志位设成 1，进入睡眠时便把它改为 0。

具体的操作取决于相应 tty_ldisc 数据结构中的函数指针 receive_buf。对于 tty_ldisc_N_TTY 而言，这个指针指向 n_tty_receive_buf()，其代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 705 778 ====================

[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()]

```
static void n_tty_receive_buf(struct tty_struct *tty, const unsigned char *cp,
                              char *fp, int count)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
    const unsigned char *p;
    char *f, flags = TTY_NORMAL;
    int  i;
    char buf[64];
    unsigned long cpuflags;

    if (!tty->read_buf)
        return;

    if (tty->real_raw) {
        spin_lock_irqsave(&tty->read_lock, cpuflags);
        i = MIN(count, MIN(N_TTY_BUF_SIZE - tty->read_cnt,
                      N_TTY_BUF_SIZE - tty->read_head));
        memcpy(tty->read_buf + tty->read_head, cp, i);
        tty->read_head = (tty->read_head + i) & (N_TTY_BUF_SIZE-1);
        tty->read_cnt += i;
        cp += i;
        count -= i;

        i = MIN(count, MIN(N_TTY_BUF_SIZE - tty->read_cnt,
                     N_TTY_BUF_SIZE - tty->read_head));
        memcpy(tty->read_buf + tty->read_head, cp, i);
        tty->read_head = (tty->read_head + i) & (N_TTY_BUF_SIZE-1);
        tty->read_cnt += i;
        spin_unlock_irqrestore(&tty->read_lock, cpuflags);
    } else {
        for (i=count, p = cp, f = fp; i; i--, p++) {
            if (f)
                flags = *f++;
            switch (flags) {
            case TTY_NORMAL:
                n_tty_receive_char(tty, *p);
                break;
            case TTY_BREAK:
                n_tty_receive_break(tty);
                break;
            case TTY_PARITY:
            case TTY_FRAME:
                n_tty_receive_parity_error(tty, *p);
                break;
            case TTY_OVERRUN:
                n_tty_receive_overrun(tty);
                break;
            default:
                printk("%s: unknown flag %d\n",
                      tty_name(tty, buf), flags);
                break;
            }
```

```
756             }
757             if (tty->driver.flush_chars)
758                 tty->driver.flush_chars(tty);
759         }
760 
761         if (!tty->icanon && (tty->read_cnt >= tty->minimum_to_wake)) {
762             kill_fasync(&tty->fasync, SIGIO, POLL_IN);
763             if (waitqueue_active(&tty->read_wait))
764                 wake_up_interruptible(&tty->read_wait);
765         }
766 
767         /*
768          * Check the remaining room for the input canonicalization
769          * mode.  We don't want to throttle the driver if we're in
770          * canonical mode and don't have a newline yet!
771          */
772         if (n_tty_receive_room(tty) < TTY_THRESHOLD_THROTTLE) {
773             /* check TTY_THROTTLED first so it indicates our state */
774             if (!test_and_set_bit(TTY_THROTTLED, &tty->flags) &&
775                tty->driver.throttle)
776                 tty->driver.throttle(tty);
777         }
778 }
```

终端的 tty_struct 结构中有 raw 和 real_raw 两个字段，都表示终端层次上的原始模式，但是 real_raw 更为原始，表示即使在链路上出现了 Break 状态，或者接收到的数据有奇偶校验出错，也都原封不动上交。具体的设置取决于所用的 termios，但可以通过系统调用 ioctl()加以改变（命令码为 TCSETSF）。不过，对于控制台终端这二者实际上没有多大区别。

可想而知，对于运行于 real_raw 模式的终端，只要把 flip 缓冲区中的内容复制到终端的 read_buf[]缓冲区中就可以了（718～732 行），否则就要通过 734～756 行的 for 循环逐个字节地边搬运边处理加工，这就是比较费时间的操作所在。

如前所述，flip 缓冲区中的代码字节都是与一个 flag 字节配对存在的，这个 flag 字节指明了代码字节的性质和类型。不过，由 tty_insert_flip_char()写入 flip 缓冲区时总是把相应的 flag 字节设成 0，就是这里的 TTY_NORMAL，所以总是调用 n_tty_receive_char()，其代码也在 drivers/char/n_tty.c 中。这个函数比较长，我们需要分段阅读。

==================== drivers/char/n_tty.c 500 552 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()]

```
static inline void n_tty_receive_char(struct tty_struct *tty, unsigned char c)
{
        if (tty->raw) {
            put_tty_queue(c, tty);
            return;
        }

        if (tty->stopped && !tty->flow_stopped &&
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
       I_IXON(tty) && I_IXANY(tty)) {
        start_tty(tty);
        return;
    }

    if (I_ISTRIP(tty))
        c &= 0x7f;
    if (I_IUCLC(tty) && L_IEXTEN(tty))
        c=tolower(c);

    if (tty->closing) {
        if (I_IXON(tty)) {
            if (c == START_CHAR(tty))
                start_tty(tty);
            else if (c == STOP_CHAR(tty))
                stop_tty(tty);
        }
        return;
    }

    /*
     * If the previous character was LNEXT, or we know that this
     * character is not one of the characters that we'll have to
     * handle specially, do shortcut processing to speed things
     * up.
     */
    if (!test_bit(c, &tty->process_char_map) || tty->lnext) {
        finish_erasing(tty);
        tty->lnext = 0;
        if (L_ECHO(tty)) {
            if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
                put_char('\a', tty); /* beep if no space */
                return;
            }
            /* Record the column of first canon char. */
            if (tty->canon_head == tty->read_head)
                tty->canon_column = tty->column;
            echo_char(c, tty);
        }
        if (I_PARMRK(tty) && c == (unsigned char) '\377')
            put_tty_queue(c, tty);
        put_tty_queue(c, tty);
        return;
    }

```

首先，如果终端运行于原始模式，就只是简单地将从 flip 缓冲区读出的字符写入终端的 read_buf[]缓冲区。

==================== drivers/char/n_tty.c 89 103 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>put_tty_queue()]

```
static inline void put_tty_queue(unsigned char c, struct tty_struct *tty)
{
	unsigned long flags;
	/*
	 *	The problem of stomping on the buffers ends here.
	 *	Why didn't anyone see this one comming? --AJK
	*/
	spin_lock_irqsave(&tty->read_lock, flags);
	if (tty->read_cnt < N_TTY_BUF_SIZE) {
		tty->read_buf[tty->read_head] = c;
		tty->read_head = (tty->read_head + 1) & (N_TTY_BUF_SIZE-1);
		tty->read_cnt++;
	}
	spin_unlock_irqrestore(&tty->read_lock, flags);
}
```

终端的 read_buf[]缓冲区是个环形缓冲区，tty->read_head 总是指向下一个可以写入的位置。如果 read_buf[]中已经写满，则丢弃新来到的输入。

对于非原始模式，或称“加工模式”，则要进行一系列的处理。这些处理包括一般的和特殊的两部分。前者包括终端的自动暂停（进入省电模式）和启动、强制转换成小写字符、往显示屏幕上“回打 ”等。后者则包括许多因具体字符而异的处理，其中也包括基于 XON/XOFF 字符（一般是 Ctrl-Q 和 Ctrl-S）的流量控制。

不过，真正要加以特殊处理的字符毕竟还是少数，try_struct 结构中的位图 process_char_map 指明了需要特殊处理的字符。只要一个字符不属于这个位图所代表的集合，对其进一步的处理就只限于“回打”，然后就通过 put_tty_queue()把该字符写入 read_buf[]缓冲区（535～550 行）。如果此前的字符是个“ erase”字符（如键盘上的 Backspace 键），则终端处于正在退字符的状态，现在新的字符既然不属于 process_char_map，就显然不再是“ erase”字符，所以先调用 finish_erasing()结束终端的退字符状态，然后就是对“回打”的处理了。

如果终端的模式设置表明需要回打，就调用 echo_char()完成这个操作。但是先要检查一下 read_buf[]缓冲区中是否还有空位可以接受当前的字符。如果已经满了就要向显示器接口写一个“ \a”字符，让它“嘟”地响一下，以免使用者误以为打入的字符已被接受。注意不要把这里调用的 put_char()与应用程序设计中的 C 库函数混淆，这是个 inline 函数，定义于 drivers/char/n_tty.c：

==================== drivers/char/n_tty.c 301 304 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>put_char()]

```
static inline void put_char(unsigned char c, struct tty_struct *tty)
{
	tty->driver.put_char(tty, c);
}
```

显然，具体的操作取决于相应 tty_driver 结构中的函数指针 put_char，对于一般的终端设备这个指针指向 tty_default_put_char()，等一下我们还要看这个函数的代码。

函数 echo_char()的代码在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 308 316 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()]

```
static void echo_char(unsigned char c, struct tty_struct *tty)
{
     if (L_ECHOCTL(tty) && iscntrl(c) && c != '\t') {
          put_char('^', tty);
          put_char(c ^ 0100, tty);
          tty->column += 2;
     } else
          opost(c, tty);
}
```

对于输入时同时按下了 Ctrl 键的字符，回打时要在该字符之前加上一个“ ^”字符。对正常输入的字符则还需要进一步的特殊处理，情况更为复杂，所以通过 opost()完成回打，其代码也在 drivers/char/n_tty.c 中：

==================== drivers/char/n_tty.c 172 233 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_try_receive_buf()>n_tty_receive_char()>echo_char()>opost()]

```
/*
 * Perform OPOST processing.  Returns -1 when the output device is
 * full and the character must be retried.
 */
static int opost(unsigned char c, struct tty_struct *tty)
{
     int  space, spaces;

     space = tty->driver.write_room(tty);
     if (!space)
          return -1;

     if (O_OPOST(tty)) {
          switch (c) {
          case '\n':
               if (O_ONLRET(tty))
                    tty->column = 0;
               if (O_ONLCR(tty)) {
                    if (space < 2)
                         return -1;
                    tty->driver.put_char(tty, '\r');
                    tty->column = 0;
               }
               tty->canon_column = tty->column;
               break;
          case '\r':
               if (O_ONOCR(tty) && tty->column == 0)
                    return 0;
               if (O_OCRNL(tty)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        c = '\n';
                        if (O_ONLRET(tty))
                                tty->canon_column = tty->column = 0;
                        break;
                }
                tty->canon_column = tty->column = 0;
                break;
        case '\t':
                spaces = 8 - (tty->column & 7);
                if (O_TABDLY(tty) == XTABS) {
                        if (space < spaces)
                                return -1;
                        tty->column += spaces;
                        tty->driver.write(tty, 0, "        ", spaces);
                        return 0;
                }
                tty->column += spaces;
                break;
        case '\b':
                if (tty->column > 0)
                        tty->column--;
                break;
        default:
                if (O_OLCUC(tty))
                        c = toupper(c);
                if (!iscntrl(c))
                        tty->column++;
                break;
        }
    }
    tty->driver.put_char(tty, c);
    return 0;
}
```

首先检查终端的写缓冲区中是否有空间。这里的特殊处理主要是针对“\r”、“\n”、“\t”以及“\b”进行的。对于“换行”键“\n”的考虑主要是自动在前面插入一个“回车”字符“\r”；对“\t”的处理则是要把它展开成若干个空格。这段程序比较简单，我们就不多说了。最后，与 put_char()一样，也是通过相应 tty_driver 结构中的函数指针 put_char 完成对终端显示器的写操作。对于一般的终端设备，这个指针指向 tty_default_put_char()，其代码还是在 drivers/char/tty_io.c 中：

==================== drivers/char/tty_io.c 1979 1985 ====================

[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()>opost()>tty_default_put_char()]

```
1979 /*
1980  * The default put_char routine if the driver did not define one.
1981  */
1982 void tty_default_put_char(struct tty_struct *tty, unsigned char ch)
1983 {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1984     tty->driver.write(tty, 0, &ch, 1);
1985 }
```

这又是通过函数指针的调用。对于控制台终端，这个指针指向 con_write()，其代码在 drivers/char/console.c 中：

==================== drivers/char/console.c 2217 2227 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()>opost()>tty_default_put_char()>con_write()]

```
2217 static int con_write(struct tty_struct * tty, int from_user,
2218             const unsigned char *buf, int count)
2219 {
2220     int  retval;
2221
2222     pm_access(pm_con);
2223     retval = do_con_write(tty, from_user, buf, count);
2224     con_flush_chars(tty);
2225
2226     return retval;
2227 }
```

注意这里的第二个参数 from_user，当这个参数为 1 时表示要写的内容来自用户空间，为 0 时则来自系统空间。指针 buf 指向要写的字符缓冲区，count 则为长度。显然，这个函数是个汇合点，一边是系统调用 read()中的回打操作，另一边则是常规的系统调用 write()。

操作的主体是 do_con_write()，也在 drivers/char/console.c 中。这个函数又比较长，我们分段阅读。

==================== drivers/char/console.c 1807 1854 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()>opost()>tty_default_put_char()>con_write()>do_con_write()]

```
1807 static int do_con_write(struct tty_struct * tty, int from_user,
1808             const unsigned char *buf, int count)
1809 {
1810 #ifdef VT_BUF_VRAM_ONLY
1811 #define FLUSH do { } while(0);
1812 #else
1813 #define FLUSH if (draw_x >= 0) { \
1814     sw->con_putcs(vc_cons[currcons].d, (u16 *)draw_from, (u16 *)draw_to-(u16 *)draw_from, y, draw_x);
\
1815     draw_x = -1; \
1816     }
1817 #endif
1818
1819     int c, tc, ok, n = 0, draw_x = -1;
1820     unsigned int currcons;
1821     unsigned long draw_from = 0, draw_to = 0;
1822     struct vt_struct *vt = (struct vt_struct *)tty->driver_data;
1823     u16 himask, charmask;
1824     const unsigned char *orig_buf = NULL;
1825     int orig_count;
```

```

	currcons = vt->vc_num;
	if (!vc_cons_allocated(currcons)) {
	    /* could this happen? */
	    static int error = 0;
	    if (!error) {
		error = 1;
		printk("con_write: tty %d not allocated\n", currcons+1);
	    }
	    return 0;
	}

	orig_buf = buf;
	orig_count = count;

	if (from_user) {
		down(&con_buf_sem);

again:
		if (count > CON_BUF_SIZE)
			count = CON_BUF_SIZE;
		if (copy_from_user(con_buf, buf, count)) {
			n = 0; /* ?? are error codes legal here ?? */
			goto out;
		}

		buf = con_buf;
	}

```

这一段程序比较简单，主要是从用户空间把输出数据复制到内核中。对于我们这个情景，输出数据本来就在内核中，所以实际上不起作用。我们继续往下看：

==================== drivers/char/console.c 1855 1913 ====================

[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()>opost()>tty_default_put_char()>con_write()>do_con_write()]

```
	/* At this point 'buf' is guarenteed to be a kernel buffer
	 * and therefore no access to userspace (and therefore sleeping)
	 * will be needed.  The con_buf_sem serializes all tty based
	 * console rendering and vcs write/read operations.  We hold
	 * the console spinlock during the entire write.
	 */

	spin_lock_irq(&console_lock);

	himask = hi_font_mask;
	charmask = himask ? 0x1ff : 0xff;

	/* undraw cursor first */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (IS_FG)
        hide_cursor(currcons);

    while (!tty->stopped && count) {
        c = *buf;
        buf++;
        n++;
        count--;

        if (utf) {
            /* Combine UTF-8 into Unicode */
            /* Incomplete characters silently ignored */
            if(c > 0x7f) {
              if (utf_count > 0 && (c & 0xc0) == 0x80) {
                    utf_char = (utf_char << 6) | (c & 0x3f);
                    utf_count--;
                    if (utf_count == 0)
                        tc = c = utf_char;
                    else continue;
              } else {
                    if ((c & 0xe0) == 0xc0) {
                        utf_count = 1;
                        utf_char = (c & 0x1f);
                    } else if ((c & 0xf0) == 0xe0) {
                        utf_count = 2;
                        utf_char = (c & 0x0f);
                    } else if ((c & 0xf8) == 0xf0) {
                        utf_count = 3;
                        utf_char = (c & 0x07);
                    } else if ((c & 0xfc) == 0xf8) {
                        utf_count = 4;
                        utf_char = (c & 0x03);
                    } else if ((c & 0xfe) == 0xfc) {
                        utf_count = 5;
                        utf_char = (c & 0x01);
                    } else
                        utf_count = 0;
                    continue;
                    }
            } else {
              tc = c;
              utf_count = 0;
            }
        } else { /* no utf */
          tc = translate[toggle_meta ? (c|0x80) : c];
        }
```

先通过 hide_cursor()把光标隐去，然后就是对输出数据的 while 循环。这里的 utf 是个宏定义，定义于 drivers/char/console_macros.h：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/char/console_macros.h 26 28 ====================

```
#define utf         (vc_cons[currcons].d->vc_utf)
#define utf_count   (vc_cons[currcons].d->vc_utf_count)
#define utf_char    (vc_cons[currcons].d->vc_utf_char)
```

就是说，如果当前（前台）虚拟控制台终端运行于 UTF-8 模式，则要把 UTF-8 代码还原成 16 位的 Unicode，最后存放在变量 tc 中。我们把这段程序留给读者与前面的 to_utf8()对照阅读，这里仅关注普通的8位代码。用于显示的字符统一为16位Unicode，一般的8位字符要通过一个数组变换成Unicode，这里的 translate 在 drivers/char/console_macros.h 中定义成当前终端的变换数组。

==================== drivers/char/console_macros.h 21 21 ====================

```
#define translate  (vc_cons[currcons].d->vc_translate)
```

每个虚拟控制台的 vc_data 数据结构中有个指针 vc_translate，指向一个大小为 256 的数组，8 位的 ASCII 或其它代码可以通过这个数组转换成 16 位 Unicode。内核中有个二维数组 translations[][]，定义于 drivers/char/consolemap.c，我们在这里只列出其中若干片断：

==================== drivers/char/consolemap.c 24 28 ====================

```
static unsigned short translations[][256] = {
  /* 8-bit Latin-1 mapped to Unicode -- trivial mapping */
  {
    0x0000, 0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007,
    0x0008, 0x0009, 0x000a, 0x000b, 0x000c, 0x000d, 0x000e, 0x000f,
. . . . . .
```

==================== drivers/char/consolemap.c 59 61 ====================

```
  },
  /* VT100 graphics mapped to Unicode */
  {
. . . . . .
```

==================== drivers/char/consolemap.c 94 98 ====================

```
  },
  /* IBM Codepage 437 mapped to Unicode */
  {
    0x0000, 0x263a, 0x263b, 0x2665, 0x2666, 0x2663, 0x2660, 0x2022,
    0x25d8, 0x25cb, 0x25d9, 0x2642, 0x2640, 0x266a, 0x266b, 0x263c,
. . . . . .
```

==================== drivers/char/consolemap.c 129 133 ====================

```
  },
  /* User mapping -- default to codes for direct font mapping */
  {
    0xf000, 0xf001, 0xf002, 0xf003, 0xf004, 0xf005, 0xf006, 0xf007,
    0xf008, 0xf009, 0xf00a, 0xf00b, 0xf00c, 0xf00d, 0xf00e, 0xf00f,
. . . . . .
```

==================== drivers/char/consolemap.c 164 165 ====================

```
  }
};
```

这个二维数组中包含着 4 个字符转换数组，每个虚拟控制台的指针 vc_translate 指向其中的一个，

可以通过系统调用 ioctl()来改变。不仅如此，还可以通过 ioctl()从用户空间下载（或上载）用户定义的字符转换数组来覆盖原有的数组，也可以下载（或上载）用户定义的字库。虚拟控制台的 ioctl()函数 vt_ioctl()是个堪称巨型的函数，其代码达 700 行之多，提供了各种各样的功能，/usr/bin 目录下的 setfont、setkeycodes、loadkeys 等工具就是建立在这个系统调用的基础上。由于篇幅的限制，我们在这里就不看这个函数了，有兴趣或需要的读者可自行阅读。对于 translations[][]中的第一个字符转换数组，即 ASCII 字符（或“ Latin-1”）的转换数组，所作的转换实际上就是保持原值不变，只不过是从 8 位变成了 16 位。从代码中可以看出，每个字符转换数组的大小是 256，所以只能支持拼音文字，而不能支持 CJKV（中日朝越）等非拼音文字。

总之，现在变量 tc 中是待显示字符的 16 位 Unicode，而变量 c 中则是该字符原来的 8 位代码。我们继续往下看（drivers/char/console.c）：

==================== drivers/char/console.c 1914 2001 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()>echo_char()>opost()>tty_default_put_char()>con_write()>do_con_write()]

```

                /* If the original code was a control character we
                 * only allow a glyph to be displayed if the code is
                 * not normally used (such as for cursor movement) or
                 * if the disp_ctrl mode has been explicitly enabled.
                 * Certain characters (as given by the CTRL_ALWAYS
                 * bitmap) are always displayed as control characters,
                 * as the console would be pretty useless without
                 * them; to display an arbitrary font position use the
                 * direct-to-font zone in UTF-8 mode.
                 */
                ok = tc && (c >= 32 ||
                           (!utf && !(((disp_ctrl ? CTRL_ALWAYS
                                        : CTRL_ACTION) >> c) & 1)))
                        && (c != 127 || disp_ctrl)
                 && (c != 128+27);

            if (vc_state == ESnormal && ok) {
                 /* Now try to find out how to display it */
                 tc = conv_uni_to_pc(vc_cons[currcons].d, tc);
                 if ( tc == -4 ) {
                             /* If we got -4 (not found) then see if we have
                               defined a replacement character (U+FFFD) */
                             tc = conv_uni_to_pc(vc_cons[currcons].d, 0xfffd);

                       /* One reason for the -4 can be that we just
                          did a clear_unimap();
                          try at least to show something. */
                       if (tc == -4)
                           tc = c;
                        } else if ( tc == -3 ) {
                             /* Bad hash table -- hope for the best */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                                        tc = c;
                                  }
                        if (tc & ~charmask)
                                        continue; /* Conversion failed */

                        if (need_wrap || decim)
                                FLUSH
                        if (need_wrap) {
                                cr(currcons);
                                lf(currcons);
                        }
                        if (decim)
                                insert_char(currcons, 1);
                        scr_writew(himask ?
                                    ((attr << 8) & ~himask) + ((tc & 0x100) ? himask : 0) + (tc & 0xff) :
                                    (attr << 8) + tc,
                                  (u16 *) pos);
                        if (DO_UPDATE && draw_x < 0) {
                                draw_x = x;
                                draw_from = pos;
                        }
                        if (x == video_num_columns - 1) {
                                need_wrap = decawm;
                                draw_to = pos+2;
                        } else {
                                x++;
                                draw_to = (pos+=2);
                        }
                        continue;
                }
                FLUSH
                do_con_trol(tty, currcons, c);
        }
        FLUSH
        spin_unlock_irq(&console_lock);

out:
        if (from_user) {
                /* If the user requested something larger than
                 * the CON_BUF_SIZE, and the tty is not stopped,
                 * keep going.
                 */
                if ((orig_count > CON_BUF_SIZE) && !tty->stopped) {
                        orig_count -= CON_BUF_SIZE;
                        orig_buf += CON_BUF_SIZE;
                        count = orig_count;
                        buf = orig_buf;
                        goto again;
                }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1995
1996            up(&con_buf_sem);
1997        }
1998
1999        return n;
2000 #undef FLUSH
2001 }
```

并不是所有的字符都可以显示，所以 1925 行根据字符转换前后的数值确定是否可以显示。如果转换前字符的数值小于 32，即“空格”，那就是控制字符，一般情况下是不能显示的。如果字符可以显示（ok 为 1），虚拟控制台又处于正常工作状态，就可以进一步处理字符的显示了。

对于虚拟控制台，系统的图形卡工作于字符模式，最终写入图形卡的是一个 16 位短字，其低字节是字符本身的代码，高字节则为字符的“属性”。字符的“属性”决定着显示该字符时的亮度、颜色等要素。所以，最后还得把字符转换成用于图形卡的 8 位代码，这种转换是由 conv_uni_to_pc()完成的，其代码在 drivers/char/consolemap.c 中：

==================== drivers/char/consolemap.c 635 667 ====================

```
int
conv_uni_to_pc(struct vc_data *conp, long ucs)
{
    int h;
    u16 **p1, *p2;
    struct uni_pagedir *p;

    /* Only 16-bit codes supported at this time */
    if (ucs > 0xffff)
        ucs = 0xfffd;      /* U+FFFD: REPLACEMENT CHARACTER */
    else if (ucs < 0x20 || ucs >= 0xfffe)
        return -1;         /* Not a printable character */
    else if (ucs == 0xfeff || (ucs >= 0x200a && ucs <= 0x200f))
        return -2;              /* Zero-width space */
    /*
     * UNI_DIRECT_BASE indicates the start of the region in the User Zone
     * which always has a 1:1 mapping to the currently loaded font.  The
     * UNI_DIRECT_MASK indicates the bit span of the region.
     */
    else if ((ucs & ~UNI_DIRECT_MASK) == UNI_DIRECT_BASE)
        return ucs & UNI_DIRECT_MASK;

    if (!*conp->vc_uni_pagedir_loc)
        return -3;

    p = (struct uni_pagedir *)*conp->vc_uni_pagedir_loc;
    if ((p1 = p->uni_pgdir[ucs >> 11]) &&
        (p2 = p1[(ucs >> 6) & 0x1f]) &&
        (h = p2[ucs & 0x3f]) < MAX_GLYPH)
        return h;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
666      return -4;        /* not found */
667  }
```

这个函数的主体是 660～664 行。虚拟控制台的 vc_data 数据结构中还有个指针 vc_uni_pagedir_loc，指向一个 uni_pagedir 数据结构，这是在 drivers/char/consolemap.c 中定义的：

==================== drivers/char/consolemap.c 174 180 ====================

```
struct uni_pagedir {
     u16       **uni_pgdir[32];
     unsigned long refcount;
     unsigned long sum;
     unsigned char *inverse_translations[4];
     int       readonly;
};
```

对于每个可显示的字符，图形卡在显示器上显示的是这个字符的特殊图形，称为“字模”（glyph）。工作于字符模式（而不是图像模式）的 VGA 图形卡可以显示 256 个字模（所以只能用于拼音文字）。根据正在使用的语言，可以将不同的字模装入图形卡，然后将在这种语言中可以显示的字符的 Unicode 变换成这些字模的编号。这样，就可以达到“本地化”的目的。当然，这只是对采用拼音文字的地区和国家而言。这个数据结构以及 conv_uni_to_pc()的作用就是实现从 Unicode 到字模编号的变换，或者说映射。对于 ASCII 码，字模的编号与字符的代码相同，但是一般而言都需要加以变换。最简单的变换当然是采用数组的方法，但是对于 16 位的 Unicode 这意味着数组的大小为 64K。这显然是没有必要的，因为对于任何一种具体的拼音文字这个数组都必定是非常稀疏的。为了提高空间效率，人们把 16 位的 Unicode 空间划分成 32 个“码页”，这里指针数组 uni_pgdir[]中的每个指针就指向一个码页，以 16 位 Unicode 编码中的最高 5 位作为下标。这样，如果对于具体的语言某个码页为全空，就不必为之分配空间，而只要使相应的指针为 0 即可。为便于管理，每个码页又划分成 32 个子页，再以次 5 位作为下标，所以 uni_pgdir[]中的指针所指向的是二维数组。这些数组的内容是在初始化时通过一个函数 con_set_default_unimap()设置的，设置的依据是一个临时性的大数组 dfont_unitable[]，初始化完成以后，这个数组所占的空间就回收另作它用了。那么，数组 dfont_unitable[]的内容又是什么，是从哪里来的呢？显然，其内容应该取决于使用的是哪一种语言。对于英语（美国英语），drivers/char 目录下有个文件 cp437.uni（表示 437 号码页的 Unicode 编码），我们在这里列出其中一些片断，让读者有个印象：

==================== drivers/char/cp437.uni 1 18 ====================

```
#
# Unicode table for IBM Codepage 437.  Note that there are many more
# substitutions that could be conceived (for example, thick-line
# graphs probably should be replaced with double-line ones, accented
# Latin characters should replaced with their nonaccented versions,
# and some upper case Greek characters could be replaced by Latin), however,
# I have limited myself to the Unicodes used by the kernel ISO 8859-1,
# DEC VT, and IBM CP 437 tables.
#
# ---------------------------------
#
# Basic IBM dingbats, some of which will never have a purpose clear
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
# to mankind
#
0x00 U+0000
0x01 U+263a
0x02 U+263b
0x03 U+2665
. . . . . .
```

==================== drivers/char/cp437.uni 47 53 ====================

```
#
# The ASCII range is identity-mapped, but some of the characters also
# have to act as substitutes, especially the upper-case characters.
#
0x20 U+0020
0x21 U+0021
0x22 U+0022 U+00a8
. . . . . .
```

==================== drivers/char/cp437.uni 84 87 ====================

```
0x41 U+0041 U+00c0 U+00c1 U+00c2 U+00c3
0x42 U+0042
0x43 U+0043 U+00a9
0x44 U+0044
. . . . . .
```

==================== drivers/char/cp437.uni 285 291 ====================

```
#
# Square bullet, non-spacing blank
# Mapping U+fffd to the square bullet means it is the substitution
# character
#
0xfe U+25a0 U+fffd
0xff U+00a0
```

例如，字模 0x42（“ B”）的 Unicode 就是 0x42；而 0x41（“ A”）则同时对应着 5 个 Unicode 代码。文件中定义的字模编号最高为 0xff，所以只定义了 256 个字模。当编译 Linux 内核时，通过一个工具 conmakehash 从这个文件为 dfont_unitable[]生成一个文件 defkeymap.c，后者则是内核源代码的一个组成部分。为帮助读者理解这一点，我们看一下 drivers/char 下面的/Makefile：

==================== drivers/char/Makefile 15 15 ====================

```
FONTMAPFILE = cp437.uni
. . . . . .
```

==================== drivers/char/Makefile 197 198 ====================

```
consolemap_deftbl.c: $(FONTMAPFILE) conmakehash
	./conmakehash $(FONTMAPFILE) > consolemap_deftbl.c
```

假定现在是在法国，要让内核显示法文，那就要提供用于法文的.uni 文件，并修改 Makefile 中 FONTMAPFILE 的定义，再编译内核就可以了。至于工具 conmakehash，其源代码 conmakehash.c 也在 drivers/char 中。

最终通过 scr_writew()写入图形卡的是一个 16 位短字，其低字节是字符本身的代码 tc，高字节则

为字符的“属性”attr。这里 scr_writew()是个宏操作，定义于 include/linux/vt_buffer.h：

==================== include/linux/vt_buffer.h 23 23 ====================

```
#define scr_writew(val, addr) (*(addr) = (val))
```

PC 机图形卡（如 VGA）上的 RAM 区间在 0xa0000 以上、BIOS 以下的位置上，其相对于起点的地址与显示屏上的位置有着一一对应的关系，CPU 可以通过访内指令直接访问。由于图形卡工作于字符方式，所以只要把字符的代码（实际上是字模的编号）连同其属性写入相应的存储单元即可，而不必关心具体的字模和像素。

显然，Linux 内核在“本地化”方面的努力只限于拼音文字，而不适用于 CJKV。以汉字为例，如果要在内核中支持汉字，图形卡就必须工作于图像方式，这样才能显示数千个常用汉字的字模。此外，即使是拼音文字，现代的图形用户界面也要求采用图像方式。不过从字符方式前进到图像方式这一步属于图像处理的范畴，我们这里就从略了。有兴趣或需要的读者可以自己钻研 drivers/video 目录下与“帧缓冲区”（frame buffer）有关的源代码。

回到 n_tty_receive_char()的代码中，我们已经看了对一般字符的回打，但是还剩下对一些特殊字符的处理，所以继续往下看（drivers/char/n_tty.c）：

==================== drivers/char/n_tty.c 553 584 ====================

[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()]

```
	if (c == '\r') {
		if (I_IGNCR(tty))
			return;
		if (I_ICRNL(tty))
			c = '\n';
	} else if (c == '\n' && I_INLCR(tty))
		c = '\r';
	if (I_IXON(tty)) {
		if (c == START_CHAR(tty)) {
			start_tty(tty);
			return;
		}
		if (c == STOP_CHAR(tty)) {
			stop_tty(tty);
			return;
		}
	}
	if (L_ISIG(tty)) {
		int signal;
		signal = SIGINT;
		if (c == INTR_CHAR(tty))
			goto send_signal;
		signal = SIGQUIT;
		if (c == QUIT_CHAR(tty))
			goto send_signal;
		signal = SIGTSTP;
		if (c == SUSP_CHAR(tty)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
send_signal:
                isig(signal, tty, 0);
                return;
            }
        }
```

这里主要是对“\r”、“\n”以及 XON/XOFF 等字符的处理。这就是说，不管这些字符是否属于 process_char_map，对这些字符总是要进行一些处理，只是方式略有不同。还有，就是几个可以导致向终端的控制进程发出信号的字符，如 Ctrl-C、Ctrl-Z 这些字符。这里还要说明，对所有这些字符的处理都是可以通过系统调用 ioctl()分别加以设置和选择的，所以 ioctl()对终端设备的驱动起着特别重要的作用。

至此，我们尚未触及对终端设备输入的“规范”（canonical）模式，下面我们就来看“规范”模式对输入的加工或者“烹调”。

==================== drivers/char/n_tty.c 585 685 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()>n_tty_receive_char()]

```
        if (tty->icanon) {
            if (c == ERASE_CHAR(tty) || c == KILL_CHAR(tty) ||
                (c == WERASE_CHAR(tty) && L_IEXTEN(tty))) {
                eraser(c, tty);
                return;
            }
            if (c == LNEXT_CHAR(tty) && L_IEXTEN(tty)) {
                tty->lnext = 1;
                if (L_ECHO(tty)) {
                    finish_erasing(tty);
                    if (L_ECHOCTL(tty)) {
                        put_char('^', tty);
                        put_char('\b', tty);
                    }
                }
                return;
            }
            if (c == REPRINT_CHAR(tty) && L_ECHO(tty) &&
                L_IEXTEN(tty)) {
                unsigned long tail = tty->canon_head;

                finish_erasing(tty);
                echo_char(c, tty);
                opost('\n', tty);
                while (tail != tty->read_head) {
                    echo_char(tty->read_buf[tail], tty);
                    tail = (tail+1) & (N_TTY_BUF_SIZE-1);
                }
                return;
            }
            if (c == '\n') {
                if (L_ECHO(tty) || L_ECHONL(tty)) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
                                put_char('\a', tty);
                                return;
                        }
                        opost('\n', tty);
                }
                goto handle_newline;
        }
        if (c == EOF_CHAR(tty)) {
                if (tty->canon_head != tty->read_head)
                        set_bit(TTY_PUSH, &tty->flags);
                c = __DISABLED_CHAR;
                goto handle_newline;
        }
        if ((c == EOL_CHAR(tty)) ||
            (c == EOL2_CHAR(tty) && L_IEXTEN(tty))) {
                /*
                 * XXX are EOL_CHAR and EOL2_CHAR echoed?!?
                 */
                if (L_ECHO(tty)) {
                        if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
                                put_char('\a', tty);
                                return;
                        }
                        /* Record the column of first canon char. */
                        if (tty->canon_head == tty->read_head)
                                tty->canon_column = tty->column;
                        echo_char(c, tty);
                }
                /*
                 * XXX does PARMRK doubling happen for
                 * EOL_CHAR and EOL2_CHAR?
                 */
                if (I_PARMRK(tty) && c == (unsigned char) '\377')
                        put_tty_queue(c, tty);

        handle_newline:
                set_bit(tty->read_head, &tty->read_flags);
                put_tty_queue(c, tty);
                tty->canon_head = tty->read_head;
                tty->canon_data++;
                kill_fasync(&tty->fasync, SIGIO, POLL_IN);
                if (waitqueue_active(&tty->read_wait))
                        wake_up_interruptible(&tty->read_wait);
                return;
        }
}

finish_erasing(tty);
```

```
666       if (L_ECHO(tty)) {
667            if (tty->read_cnt >= N_TTY_BUF_SIZE-1) {
668                 put_char('\a', tty); /* beep if no space */
669                 return;
670            }
671            if (c == '\n')
672                 opost('\n', tty);
673            else {
674                 /* Record the column of first canon char. */
675                 if (tty->canon_head == tty->read_head)
676                      tty->canon_column = tty->column;
677                 echo_char(c, tty);
678            }
679       }
680
681       if (I_PARMRK(tty) && c == (unsigned char) '\377')
682            put_tty_queue(c, tty);
683
684       put_tty_queue(c, tty);
685  }
```

我们不对这些代码详加解释了，读者现在应该已经有了这种能力。这里我们只是要强调，在“规范”模式下，尽管终端设备的 read_buf[]缓冲区中已经有了数据，却并不马上就唤醒可能正在睡眠中等待着要从该终端读出的进程，而要到接收到“\n”字符（有可能从“\r”转换而来，见 557 行）以后，或者接收到 EOF、EOL 等字符时才来唤醒（见 660 行）。相比之下，在非“规范”模式下，包括原始模式下，则只要缓冲区中的字节数量达到预先设定的 minimum_to_wake，就会唤醒正在睡眠等待的进程，而 minimum_to_wake 一般是 1。具体地，这发生于从 n_tty_receive_char()返回到 n_tty_receive_buf()以后，为便于阅读，我们把有关的几行代码再列出于下（drivers/char/n_tty.c）。

==================== drivers/char/n_tty.c 761 765 ====================
[console_softint()>run_task_queue()>flush_to_ldisc()>n_tty_receive_buf()]

```
761       if (!tty->icanon && (tty->read_cnt >= tty->minimum_to_wake)) {
762            kill_fasync(&tty->fasync, SIGIO, POLL_IN);
763            if (waitqueue_active(&tty->read_wait))
764                 wake_up_interruptible(&tty->read_wait);
765       }
```

这里的 762 行用于异步 I/O，读者可参阅“系统调用 select()与异步 I/O 一节”。

读者在前面看到，不光控制台有个 tasklet，键盘本身也有个 bh 函数 kbd_bh()，不过这个 bh 函数处理的只是键盘上的发光二极管，所以对于键盘的读操作没有什么影响。这个 bh 函数的代码在 drivers/char/keyboard.c 中，我们就不看了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

8.9　通用串行外部总线 USB

8.9.1 USB 总线简介

USB（Universal Serial Bus）总线是 20 世纪 90 年代发展起来的一种“通用串行外部总线”，目前在使用中的基本上是 1998 年公布的 USB 1.1 版。最新的版本是 USB 2.0，目前已有芯片投入市场，但是有关的产品（设备）还不多，而支持 2.0 版的设备驱动程序（以及相应版本的操作系统）则一般还在开发或试运行阶段。

首先，USB 是一种“总线”。与传统的外部设备与主机之间的连接方式不同，它允许将不同种类的外部设备混合连接到同一个接口上。可是，它又与计算机内部的总线（如 PCI 总线）不同，CPU 不能通过访内指令或 I/O 指令直接访问连接在 USB 上的设备，而要通过一个“ USB 控制器”，间接地与连接在 USB 上的设备打交道，USB 总线存在于计算机的外部，所以说是外部总线。还有，USB 的信号线一共只有两条（外加两条电源线），线上的信号是串行的，所以是“串行外部总线”。至于说“通用”，那是因为 USB 总线的设计从一开始就考虑到了许多不同种类的外部设备，从低速的键盘和鼠标等“人机交互设备（HID）”，到速度较高的通信设备（如 Modem）和存储设备（如 CDROM），乃至摄像机和显示器等多媒体设备，只要带有 USB 接口就都可以连接到 USB 总线上，并且可以在计算机带电的条件下“现插现用”（plug and play）。

在传统的计算机系统结构中，接口卡可以看作是相应外部设备的一部分，就好像是外部设备派驻在主机内部的联络员一样。所以，每一个接口一般都只能连接到同一种的设备。虽然通过所谓“菊花链”（daisy chain）方式可以把若干同种设备连接到同一接口上，却不能将不同种类的设备混合连接到同一接口上。显而易见，这种结构的可扩充性很差，因为可以插入主机的接口卡数量总是有限的。事实上，应用中很早就有了在同一接口上混合连接不同设备的要求。例如在并行口上既连接打印机又连接扫描仪、可能还要再连上一台“ Zip 驱动器”，就反映了这样的要求。而且，即使主机中有足够多的插槽，也还是有很多问题，例如中断向量的分配和管理、接口卡的成本等等，甚至因太多的连接电缆挤在一起而造成的困难也是个问题。特别值得一提的是：要插入一块接口卡通常得要打开机盖并且关闭电源，而不能在系统加电的情况下随意增减设备。在这样的条件下，所谓“现插现用”实际上是很难真正成为现实的。随着计算机应用的日益普及，这些问题就愈来愈突出了。

另一方面，更为重要的是：电话设备与计算机的结合以及多媒体技术的发展，也对外部设备的连接与驱动提出了新的要求。在电话设备（以及多媒体设备）中普遍采用所谓“等时”（isochronous）传输方式，通常将一个容量较大的信道按时间划分成若干较小的音频或视频信道。例如“ T1”传输线就是将一个 1.54Mb 的信道按时间划分成 8000 个 24 字节的“帧”（frame），而每个帧的 24 个字节又分别用于 24 个 64Kb 的数字化话音信道，使每个话音信道在每 125 微秒的时间内就能传送一个字节。与传统的传输方式相比，等时传输有两大特点。第一，等时传输有比较严格的时间要求，必须为每个信道维持均匀的流量。例如，如果对音频信号的传输忽快忽慢，那么重放出来的声音就会变成有如磁带录音机转速不稳时那样的怪声怪调。相比之下，传统的计算机外部设备（例如从磁盘读入文件时）则并没有这样的要求。第二，等时传输对误码的要求不高，少量的误码（甚至丢失少量数据）对于音频表

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

现为噪音，对视频则表现为画面上的一些“雪花点”，都还是可以容忍的。所以等时传输不需要有 CRC 校验、奇偶校验一类的检错手段。其实，那怕明知传输中有错，甚至丢失，也不能要求对方重发，因为由重发而造成的延迟很可能反而更不能容忍。反之，传统的计算机外部设备对误码的要求就很高，所以通常都带有检错的手段，如果发现出错就要重发。不过，实际上等时传输总是有一定的缓冲，所以只要是在一定的限度内，时间上略有飘移也是允许的。总之，等时传输基本上是周期性的，但是又并非严格意义上的周期信号，并且收发双方也并不需要严格意义上的同步。例如，在上面讲到的“ T1 ”传输线上，接收方只需知道哪一点是一个帧的起点，却不需要知道到底是哪一个帧的起点，因为相差 10 个帧也不过是 1.25 毫秒，通话的双方都感觉不到。一个固定而不太大的延迟是允许的（就像越洋电话那样）。这样的传输方式就称为等时传输，这并不是新技术，但是以前的计算机外部设备中没有这方面的需求。而在 20 世纪 90 年代初，则由于计算机与电话技术相结合的趋向以及多媒体技术的发展，而使之提到了日程上。以电话设备为例，就需要既有支持容量较大的等时方式传输（用于话音），又支持报文传递（用于控制）的外设接口。同时，这种外设接口仍然要适合于传统的外部设备。

USB 正是在这样的背景下发展起来的。90 年代初，人们已经在设备驱动、数据通信、局部网络以及大规模集成电路等方面积累了丰富的经验，USB 的设计从各方面都吸取了营养，而大规模集成电路技术则为 USB 的实现奠定了物质基础。从 90 年代中期开始生产的 PC 机几乎毫无例外地全都带有 USB 接口。

从功能上讲，主机的 USB 接口既可以通过 USB 电缆直接连接到一台支持 USB 的外部设备（我们称之为“ USB 设备”），也可以先连接到一个“ USB 集中器”（USB Hub），再由集中器分叉连接到其它集中器或外部设备，从而形成一种星形结构。每根 USB 电缆的长度是 16 尺（约 5 米），通过集中器级连时最多可以穿越 5 个集中器，从而使最大半径达到 96 尺（约 29 米）。USB 上的信号传输速度有两种，一种是“低速”（Low Speed），为每秒 1.5 兆位，用于外接键盘、鼠标器等低速设备（常称为 HID，即“人机交互设备”）；另一种是“全速”（Full Speed），为每秒 12 兆位（作为比较，Ethernet 的速度是每秒 10 兆位），用于一般的外部设备，包括音频和视频；两种速度的设备可以混用。在后来公布的 USB 标准 2.0 版中，在此基础上又增加了一种传输速度，称为“高速”（High Speed），达到每秒 480 兆位。USB 设备可以在系统加电的条件下自由地插上或拔下，称为“热插入”。而且，对耗电量很小的设备或集中器还可以由主机通过 USB 电缆供电。

凡是支持 USB 的外部设备，即 USB 设备（集中器是一种特殊的 USB 设备），都带有 USB 通信控制器，里面实际上包括了一个微处理器。由此可见，USB 的实现和推广在大规模集成电路技术还不发达、微处理器还比较贵的时代是不现实的。由于主机和 USB 设备都带有控制器，而主机中的控制器起着核心的作用。为了防止混淆，一般把主机一方的控制器称为“主控制器”（Host Controller，常缩写为 HC）。不过，在本书中并不关心设备一方的操作，所以讲到“ USB 控制器”时总是指主控制器。除主控制器外，每条 USB 总线一定有个“根集中器”，通常与主控制器集成在同一芯片中。

另一方面，由此也可看出，在某种意义上也可以把 USB 看成是一种计算机网络，一种主 / 从结构的星形网络。之所以说是主 / 从结构，是因为信息在 USB 上的传输只能由主机启动，而不能由设备启动，设备永远处于被动的地位。

通过以上的简介，读者已可了解：对于主机系统，USB（实际上是 USB 控制器）本身也是一项设备，但是对 USB 操作本身并不是终极的目的，而是为了通过 USB 对具体的外部设备进行操作。换言之，对 USB 的操作是手段而不是目的。然而，在另一方面，这种手段却有可能比目的更复杂，因此有

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

必要特别加以介绍。

USB 的信息传输方式是比较特殊的（具有网络和数据通信背景的读者则可能会觉得很自然）。

首先，USB 通过具有一定格式的“信包”（packet）按一定的“规程”（protocol）传输信息，并根据信息（内容）的性质分成如下 4 种传输类型：

(1) 控制型（Control）。主要用于设备的“配置”与控制。信包的容量可以是 8、16、32 或 64 字节，取决于具体的设备（低速设备只支持 8 字节）。在 USB 的整个带宽（传输能力）中，有 10%是为这种信息保留的。这就是说，只要有足够多的控制型信息等着要传递，就必须保证有 10%的带宽用于这些信息，其它信息再多也不能把这部分带宽给挤掉。但是，如果没有足够多的控制型信息要传递，则可以把这部分带宽用于其它类型的信息。控制型信包的传递是带有检错、并须由接收方加以确认的“可靠”传递，如果发现传输出错就要重发。

(2) 等时型（isochronous）。主要用于实时的音频和视频信号。这种信息是周期性的，又是实时的，对信息传递是否及时有很高的要求，但是对误码却比较能容忍。所以，保证用于等时型信息的带宽是很重要的。USB 为等时型信息和中断型信息（也是周期性的）保留 90%的带宽（如果有足够多的周期性信息要传递）。另一方面，等时型信息的传递不带检错，也不需要确认，因而就不存在重发的问题。等时型传输的信包通常较大，最高可达 1023 字节。

(3) 中断型（Interrupt）。名曰“中断”型，实际上却是用于对 USB 设备的周期性查恂。USB 设备不存在主动向主机发送“中断请求”的能力，只能被动地接受主机通过 USB 总线查询。中断型信息的传递既有时间上的要求，又必须是可靠传递，但是信包较小。信包大小与控制型信包相同。中断型信息和等时型信息二者合在一起不能超过 USB 总线带宽的 90%。

(4) 成块型（Bulk）。用于信息量相对较大，没有很强的时间要求，但是要求可靠传递（带有检错，接收方须确认）的信息。对成块型信息的传递在时间上是没有保证的，USB 总线不为成块型信息保留带宽，只是在执行了前三种传输以后还有时间剩余时才来执行成块型传输。信包的容量取决于具体的设备，但最大不超过 1023 字节。

同时，USB 控制器把总线上的时间划分成同定大小的“ frame”，即“框架”（我们把 frame 这个词翻译成“框架”而不是“帧”，一来与网络技术中的“帧”有所区别，二来意义上也似乎更为贴切）。每个框架的大小是 1 毫秒。在每个框架中主机都可以与 USB 总线上的设备相互发送许多信包，不过在每一特定的时刻只能在一个方向上发送，所以是“半双工”链路。主机与设备之间的通信只能由主机启动，而设备则永远处于从属的、被动的地位。因而主机与设备间是主 / 从关系。在每一个框架内可以有许多个主机与设备间的“交互”（Transaction），即二者间若干信包的交换。每个信包传递的方向可以是从主机到设备，也可以是从设备到主机，但是每个交互中的第一个信包总是从主机到设备，因为只有主机可以发起一次交互。信包的大小因传输方式和设备类型而异，对于全速设备的控制（型）传输最大为 64 字节，对低速设备则不超过 8 个字字节，但是等时（型）传输的信包可达 1023 字节。每个框架的开头总是由主机向总线上的所有设备广播一个特殊的“框架开始”（SOF）信包，作为一种同步手段。框架的划分是硬性的、没有弹性的，时候一到，主机的 USB 控制器就发出 SOF 信包。

主机与 USB 总线上各设备之间的位同步，靠调制在信号波形中的时钟脉冲实现，框架的同步靠 SOF 信包实现，而信包的同步则靠信包头部的特殊格式实现。

一般而言，主机与设备之间的信息传递又可以根据其目的分成两种。一种是应用信息的传递，包括数据以及由具体设备规定的一些应用层上的控制 / 状态信息，例如待打印的字符串，又如对扫描器

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的“开启光源”命令、“开始扫描”命令等等，就属于这一种。从传输的角度看，这些信息对于 USB 总线是“透明”的，USB 总线只是把这些信息以信包的形式作为无格式字节流传递给对方，而并不关心其内容，也不规定其格式。当然，应用软件以及具体设备的驱动程序可以自己规定这些信息的格式。

另一种是 USB 总线为维持其本身的正常运行和管理所需的一些“ USB 层”的控制 / 状态信息，例如，为要了解某一设备是否还在总线上运行，或者要了解某一设备都具有哪些功能，就需要从设备读入其作为 USB 设备必须提供的状态信息，等等。我们有时称此种控制 / 状态信息为“高层”控制 / 状态信息，之所以称为“高层”是为了与前面所说的低层控制 / 状态信息相区别。USB 总线为这些信息规定了特定的格式，例如后面要讲到的设备描述体、接口描述体等等。不过，根据具体的情况，在传递控制 / 状态信息时往往也可以附加传递一些应用信息。

本来，USB 总线上信息传输的目的就在于传递这些应用信息和“高层”控制 / 状态信息。可是，为了启动传递这些信息以及保证这些信息得到可靠的传递，还需要传递一些附加的信息。这些信息的传递本身并不是目的，而只是为达到上述目的而采取的手段。

每次高层控制 / 状态信息或者（以及）应用信息的传递称为一次“传输”（transfer），因信息类型的不同可以分成上述控制、中断、等时以及成块 4 种。同时，按传输的方向又可分成输入和输出两种。每次传输都由一次或数次“交互”构成。每次交互又包括三个信包（等时传输的交互例外、只有两个信包）的传递，其中第一个信包总是由主机发出的低层控制信息，称为“传令”（token）信包，信包的内容表明了交互的对象以及后续信包的传递方向；然后是一个载运着应用信息或高层控制 / 状态信息的信包，称为“数据”（data）信包；最后则是由数据信包的接收方发出载运着确认信息的信包，称为“握手”（handshake）信包。不过，在等时传输方式的交互中接收方不发出握手信包，所以只有两个信包。可见，传令信包和握手信包所载运的信息是为完成交互所需的“低层”控制 / 状态信息，在性质上有别于上述的高层控制 / 状态信息。高层控制 / 状态信息的目的在于维持 USB 总线的正常运行和管理，而低层控制 / 状态信息的目的在于保证应用信息或高层控制 / 状态信息的正确传递。每个交互只传递一个数据信包，数据信包传递的方向即为交互的方向，往往也就是传输的方向。交互是一个不可分割的整体，其三个（或两个）信包的传递必须在同一框架中完成。在主机发出传令信包，启动了一次交互以后，目标设备必须立即（在规定的时间内）作出反应，或从主机接收数据信包，或将数据信包发送给主机，否则本次交互便失败了。而一次传输，如果包含多次交互，则可以跨越多个时间框架。在前述的 4 种传输方式中，成块、等时和中断三种应用信息的传输都只包含一种（虽然可能多个）交互，但是控制（型）传输则一般需要通过三种不同性质的交互才能完成，称为控制传输的三个“阶段”。其中第一个阶段为“ SETUP”阶段，只有一个交互，在这个交互中传递的是高层的控制命令，如“读配置信息”、“设置地址”等等。第二个阶段是数据阶段，根据需要可以有多个交互，也可以没有。在这些交互中传递的是与本次命令相联系的信息，也可以是一些应用信息；如果不需要传递这样的信息，则也可以跳过数据阶段。第三个阶段是状态阶段，这个交互中传递的是与操作目标有关的状态信息。相比之下，成块、等时和中断三种传输都只有一个阶段，即数据阶段。

总之，USB 总线上的信息流通都以信包的形式进行。按照信包的性质和作用，可以分成下列几类：

(1) 传令（Token）。由主机发出，用来启动一次交互。除本次交互目标的地址外，信包中还含有交互的方向和性质，IN 表示输入，OUT 表示输出，SETUP 则用于控制传输的启动。

(2) 数据。根据交互的方向，由主机或设备发出，其内容为应用信息或高层控制 / 状态信息。

(3) 握手。由数据信包或传令信包的接收方（或集中器）发出，说明对数据信包或传令信包的接

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

收情况，ACK 表示成功，NAK 表示出错或无反应，STALL（由集中器发出）表示不能接收。

(4) SOF。用于框架的分隔，信包中含有框架的序号。

(5) 低速设备前缀。低速设备在每个信包之前都要加上一个特殊的前缀。

每个物理的 USB 设备上可以有一个或多个“功能”，相当于“逻辑设备”。在有些设备中，这些功能的划分和组合是可以改变的，此时一种特定的划分和组合就称为一种“配置”（configuration）。在这方面，电话通信设备就是很典型的。以前述的 T1 设备为例（中国采用 E1，有 32 个 64Kb 数字化信道），其 24 个数字化信道就可以按需要加以组合，例如将其中的 12 个信道用于电话，而拿出 8 个信道组合成一个 64KB 的信道用于互联网，剩下 4 个信道则用作通向 4 个营业所的专线连接，这就是一个具体的配置。同时，为了与主机通信的需要，在 USB 设备中设置了一些“端点”（endpoint）。每个端点只支持一种性质的传输。除用于控制传输的端点为整个设备所公用外，其它端点都属于某个具体的功能，即逻辑设备。反过来，根据具体的需要，每个功能可以有多个通信端点。在某些设备中，这些通信端点的从属就像功能的划分一样是可以改变的。

主机与具体 USB 设备的联系在概念上按传输类型和对象的不同而分成许多“管道”（Pipe）。如前所述，每个 USB 设备可以有若干个“功能”，每个功能可以提供若干通信“端口”（Port），而主机与每个端口之间就是一个逻辑上的管道（注意不要与进程间通信管道混淆）。

主机负有调度 USB 总线上信息传输的责任。首先是把系统中的等时传输流量分配给各个时间框架。所以，每个等时传输中的各个交互明确地属于某个时间框架。每条 USB 总线有 1024 个时间框架，相应地就有 1024 个等时交互队列。队列中的每个数据结构（称为“交互描述块”）都代表着一个属于等时传输的交互请求。USB 总线控制器每一秒钟扫描一遍所有这些队列。所以，每个队列在每一秒钟的时间内得到一次执行。其余的三种传输请求都不分配到具体的时间框架，而是根据传输的类型排在各目的队列中。USB 总线控制器在执行完每个框架中的等时交互以后就会来执行这些队列中的交互（请求）。不过中断传输（只含一个交互）是以交互为单位挂入队列，而控制传输和成块传输则以传输为单位挂入队列，每个控制传输或成块传输本身又是一个交互请求的队列（即使队列中只有一个交互）。

完成了对各种交互请求的调度，建立起相应的数据结构和队列以后，CPU 就完成了任务，以后就是 USB 总线控制器的事了。在运行中，USB 总线控制器根据其内部对时钟脉冲的计数确定在什么时候开始一个框架以及哪一个框架，接着就“执行”该框架的等时交互队列。然后，执行完一个框架的等时交互以后，还要执行若干中断交互。到执行完一个框架的等时交互和中断交互时，应该至少还有 10% 的时间剩余，此时就来执行其它队列，一直到框架中剩余的时间已不足以完成一次交互、而只好停下来等待下一个框架为止。对于控制传输和成块传输的交互请求队列（也就是对每个控制传输或成块传输），执行时有两种不同的方式。一种称为“纵向执行”，就是先顺着一个队列执行完所有的交互请求，再开始另一个队列；另一种称为“横向执行”，就是在一个队列中执行一个（或几个）交互请求，然后就转到下一个队列中。等时交互和中断交互在执行以后仍留在队列中，而控制交互和成块交互则在执行以后由 USB 总线控制器自动予以脱链。CPU 仅在需要改变调度或处理传输结果时才需要介入。

可见，USB 上的信息传输，尤其是主机一侧的过程，是个相当复杂、并且时间性要求很高的过程，需要硬件和软件分工合作才能完成。那么怎样分工呢？这就需要在硬件和软件之间划出一个标准的“主机控制器界面”。目前有两种这样的界面。一种是由 Intel 制定的，称为“Universal Host Controller Interface”，缩写为 UHCI；另一种是由 Compaq、Microsoft 以及 National Semiconductor 联合制定的，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

称为“ Open Host Controller Interface”，缩写为 OHCI。大体上说，OHCI 把更多的功能划到硬件一边，从而相应的芯片就更复杂一些，软件则可以简单一些；而 UHCI 则让软件方面多承担一些，对硬件的要求就相对低一些。不言而喻，控制器芯片的不同当然会导致驱动软件的不同，但是这种不同只存在于驱动软件的低层。所以，USB 总线的驱动程序又分成两层，其中“ USB 层”是共同的；在它下面则是“ HC”层，根据所用的芯片而选用 UHCI 或 OHCI 驱动程序。Linux 内核对二者都提供支持，可以通过条件编译选择项选用。对 OHCI 的实现在 drivers/usb/usb-ohci.c 和 drivers/usb/usb-ohci.h 两个文件中。对 UHCI 的实现则又有两种，也是通过条件编译选择项选用。其中之一在 drivers/usb/usb-uhci.c 和 drivers/usb/usb-uhci.h 两个文件中，另一种则在 drivers/usb/uhci.c 和 drivers/usb/uhci.h 两个文件中。我们在本节中阅读代码时采用后者，但是读者席该不难学一反三。我们之所以选用 UHCI，是因为 UHCI 的软件更复杂一些，但是读了以后对 USB 的理解会更深一些。目前，Linux 内核 2.4.0 版所实现的是 USB 1.1 版，以后肯定很快就会支持 USB 2.0 版。

除 USB 以外，大约在同一时期还发展起了另一种类似的串行外设总线，称为“ Firewire”。后来由 IEEE 加以标准化，成了 IEEE1394 标准，Linux 内核也支持 IEEE1394。但是，从市场的情况来看，USB 已经得到更为广泛的应用，并已进入良性循环，所以也更为重要，而且二者在逻辑上颇为相似。本书既然花了较大的篇幅介绍 USB，就不再涉及 IEEE1394 了。内核中与此有关的代码基本上都在 drivers/ieee1394 下面，有兴趣或需要的读者可以自己加以研究。

此外，还要提一下并行外设总线。原先的“并行口”也已发展成为并行外设总线，并且也已由 IEEE 加以标准化，成为了 IEEE1284 标准；同样，Linux 内核也支持 IEEE1284。相比之下，IEEE1284 比 USB 简单，功能也没有 USB 强，使用也没有 USB 方便，估计可能会慢慢被 USB 取代。限于本书的篇幅，我们也不能在此介绍 IEEE1284 了。与此有关的几个文件在 drivers/parport 下面，读者不妨自己阅读。

从串行口和并行口朝着串行外设总线和并行外设总线的发展过程中，我们可以看出在传统的设备驱动层中正在形成一个新的子层。使用这些总线以后，原来的“底层”驱动程序现在要依靠相应总线驱动程序所提供的服务才能访问目标设备了。似乎可以说：内核在设备驱动方面的重点正在从提供直接的设备驱动向提供实现设备驱动的手段和环境转移。

8.9.2 USB 总线的初始化和 USB 设备的枚举

我们先看 USB 总线本身的初始化。首先，USB 控制器（连同根集中器）连接在 PCI 总线上，是一个 PCI 设备，在 PCI 总线的初始化过程中也会受到枚举。PCI 设备的初始化完成以后，在 PCI 总线树中就有了代表着具体 USB 总线控制器的 pci_dev 数据结构，并已为控制器的 I/O 区间和 RAM 区间分配和设置了总线地址。

在 USB 总线控制器的设备驱动程序方面，则要为其准备下一个 pci_driver 数据结构，其类型定义在 include/linux/pci.h 中：

==================== include/linux/pci.h 449 457 ====================

```
449  struct pci_driver {
450       struct list_head node;
451       char *name;
452       const struct pci_device_id *id_table;/* NULL if wants all devices */
```

```
453     int (*probe)(struct pci_dev *dev, const struct pci_device_id *id);    /* New device inserted */
454     void (*remove)(struct pci_dev *dev); /* Device removed (NULL if not a hot-plug capable driver) */
455     void (*suspend)(struct pci_dev *dev);/* Device suspended */
456     void (*resume)(struct pci_dev *dev); /* Device woken up */
457 };
```

这个数据结构为通用的 PCI 设备管理机制提供了几个函数指针，特别是为一个通用的、一般化的 PCI 设备初始化过程提供了函数指针 probe。供这个 PCI 设备的初始化过程“回叫”，以完成具体设备的初始化。对于遵循 UHCI 界面的 USB 控制器，其 pci_driver 数据结构为 uhci_pci_driver，定义于 drivers/usb/uhci.c：

==================== drivers/usb/uhci.c 2458 2469 ====================

```
2458 static struct pci_driver uhci_pci_driver = {
2459     name:          "usb-uhci",
2460     id_table:&uhci_pci_ids [0],
2461
2462     probe:         uhci_pci_probe,
2463     remove:        uhci_pci_remove,
2464
2465 #ifdef    CONFIG_PM
2466     suspend:  uhci_pci_suspend,
2467     resume:        uhci_pci_resume,
2468 #endif    /* PM */
2469 };
```

结构中的指针 id_table 应该指向一个 pci_device_id 结构数组，表明由这个数据结构所确定的设备驱动程序适用于哪一些 PCI 设备。对此，drivers/usb/uhci.c 中相应地定义了数组 uhci_pci_ids[]：

==================== drivers/usb/uhci.c 2441 2454 ====================

```
2441 static const struct pci_device_id __devinitdata uhci_pci_ids [] = { {
2442
2443     /* handle any USB UHCI controller */
2444     class:         ((PCI_CLASS_SERIAL_USB << 8) | 0x00),
2445     class_mask:    ~0,
2446
2447     /* no matter who makes it */
2448     vendor:        PCI_ANY_ID,
2449     device:        PCI_ANY_ID,
2450     subvendor:     PCI_ANY_ID,
2451     subdevice:     PCI_ANY_ID,
2452
2453     }, { /* end: all zeroes */ }
2454 };
```

从其定义可以看出，它适用于所有类型为 PCI_CLASS_SERIAL_USB，子类型为 0 的 PCI 设备，即 USB 控制器，而不管是由哪一家厂商提供。

准备好这些数据结构以后，就可以通过 inline 函数 pci_module_init()向系统登记具体的设备驱动程

序，并对设备进行初始化。其代码在 include/linux/pci.h 中：

==================== include/linux/pci.h 602 629 ====================

```
602 /*
603  * a helper function which helps ensure correct pci_driver
604  * setup and cleanup for commonly-encountered hotplug/modular cases
605  *
606  * This MUST stay in a header, as it checks for -DMODULE
607  */
608 static inline int pci_module_init(struct pci_driver *drv)
609 {
610     int rc = pci_register_driver (drv);
611 
612     if (rc > 0)
613         return 0;
614 
615     /* iff CONFIG_HOTPLUG and built into kernel, we should
616      * leave the driver around for future hotplug events.
617      * For the module case, a hotplug daemon of some sort
618      * should load a module in response to an insert event. */
619 #if defined(CONFIG_HOTPLUG) && !defined(MODULE)
620     if (rc == 0)
621         return 0;
622 #endif
623 
624     /* if we get here, we need to clean up pci driver instance
625      * and return some sort of error */
626     pci_unregister_driver (drv);
627 
628     return -ENODEV;
629 }
```

这里通过 pci_register_driver()完成 PCI 设备驱动程序的登记和初始化，这就是前面所讲的通用、一般化的 PCI 设备初始化过程。这个函数返回一个计数，表示在 PCI 总线上找到了几个这样的设备。一般，如果这个函数返回 0 就要调用 pci_unregister_driver()撤销登记，但是如果 PCI 总线允许“热插入”。即在加电以后的运行过程中带电插入设备，而驱动程序又并非通过可安装模块实现，则不应撤销登记（621 行）；冈为以后热插入此种设备时仍需要执行这个驱动程序，应该保留着，以备热插入此种设备之需。函数 pci_register_driver()的代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 324 336 ====================

[pci_module_init()>pci_register_driver()]

```
int
pci_register_driver(struct pci_driver *drv)
{
    struct pci_dev *dev;
    int count = 0;

    list_add_tail(&drv->node, &pci_drivers);
```

```
    pci_for_each_dev(dev) {
        if (!pci_dev_driver(dev))
            count += pci_announce_device(drv, dev);
    }
    return count;
}
```

首先将 USB 总线控制器的 pci_driver 数据结构链入内核中的队列 pci_drivers，这就是所谓“登记”。然后通过一个循环对所有的 pci_dev 数据结构调用 pci_dev_driver()，统计一下有多个 PCI 设备尚未与具体的驱动程序挂上钩。这个函数的代码在 drivers/pci/pci.c 中：

==================== drivers/pci/pci.c 456 468 ====================
[pci_module_init()>pci_register_driver()>pci_dev_driver()]

```
struct pci_driver *
pci_dev_driver(const struct pci_dev *dev)
{
    if (dev->driver)
        return dev->driver;
    else {
        int i;
        for(i=0; i<=PCI_ROM_RESOURCE; i++)
            if (dev->resource[i].flags & IORESOURCE_BUSY)
                return &pci_compat_driver;
    }
    return NULL;
}
```

如果一个设备的 pci_dev 结构尚未与任何驱动程序挂钩，并且其所有地址区间都尚未启用，则 pci_dev_driver()返回 0。这样的 pci_dev 结构需要通过 pci_announce_device()加以比对（drivers/pci/pci.c）。

==================== drivers/pci/pci.c 299 322 ====================
[pci_module_init()>pci_register_driver()>pci_announce_device()]

```
static int
pci_announce_device(struct pci_driver *drv, struct pci_dev *dev)
{
    const struct pci_device_id *id;
    int ret = 0;

    if (drv->id_table) {
        id = pci_match_device(drv->id_table, dev);
        if (!id) {
            ret = 0;
            goto out;
        }
    } else
        id = NULL;

    dev_probe_lock();
    if (drv->probe(dev, id) >= 0) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            dev->driver = drv;
            ret = 1;
        }
        dev_probe_unlock();
out:
        return ret;
}
```

可想而知，所谓比对是将具体设备的类型、厂家等等在 PCI 枚举阶段从设备收集的信息与 USB 驱动程序的数组 uhci_pci_ids[]进行比对，具体是由 pci_match_device()完成的（drivers/pci/pci.c）：

===================== drivers/pci/pci.c 284 297 =====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>pci_match_device()]

```
const struct pci_device_id *
pci_match_device(const struct pci_device_id *ids, const struct pci_dev *dev)
{
        while (ids->vendor || ids->subvendor || ids->class_mask) {
            if ((ids->vendor == PCI_ANY_ID || ids->vendor == dev->vendor) &&
                (ids->device == PCI_ANY_ID || ids->device == dev->device) &&
                (ids->subvendor == PCI_ANY_ID || ids->subvendor == dev->subsystem_vendor) &&
                (ids->subdevice == PCI_ANY_ID || ids->subdevice == dev->subsystem_device) &&
                !((ids->class ^ dev->class) & ids->class_mask))
                 return ids;
            ids++;
        }
        return NULL;
}
```

如果比对结果相符，就找到了一个 USB 总线控制器，此时便通过驱动程序提供的函数指针 probe 对其进行初始化。对于遵循 UHCI 界面的 USB 总线控制器，这个函数是 uhci_pci_probe()，其代码在 drivers/usb/uhci.c 中：

===================== drivers/usb/uhci.c 2376 2406 =====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()]

```
static int __devinit uhci_pci_probe(struct pci_dev *dev, const struct pci_device_id *id)
{
        int i;

        /* disable legacy emulation */
        pci_write_config_word(dev, USBLEGSUP, 0);

        if (pci_enable_device(dev) < 0)
            return -ENODEV;

        if (!dev->irq) {
            err("found UHCI device with no IRQ assigned. check BIOS settings!");
            return -ENODEV;
        }

```

```
    /* Search for the IO base address.. */
    for (i = 0; i < 6; i++) {
        unsigned int io_addr = pci_resource_start(dev, i);
        unsigned int io_size = pci_resource_len(dev, i);

        /* IO address? */
        if (!(pci_resource_flags(dev, i) & IORESOURCE_IO))
            continue;

        /* Is it already in use? */
        if (check_region(io_addr, io_size))
            break;

        pci_set_master(dev);
        return setup_uhci(dev, dev->irq, io_addr, io_size);
    }
```

首先通过 pci_enable_device()及其下面的一系列子程序启用作为 PCI 设备的 USB 控制器。这些函数都在 drivers/pci/pci.c 或 arch/i386/kernel/pci-pc.c，arch/i386/kernel/pci-i386.c 中，我们把它们留给读者自己阅读，作为对 PCI 总线一节的复习。

==================== drivers/pci/pci.c 242 259 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>pci_enable_device()]

```
/**
 * pci_enable_device - Initialize device before it's used by a driver.
 * @dev: PCI device to be initialized
 *
 *  Initialize device before it's used by a driver. Ask low-level code
 *  to enable I/O and memory. Wake up the device if it was suspended.
 *  Beware, this function can fail.
 */
int
pci_enable_device(struct pci_dev *dev)
{
    int err;

    if ((err = pcibios_enable_device(dev)) < 0)
        return err;
    pci_set_power_state(dev, 0);
    return 0;
}
```

==================== arch/i386/kernel/pci-pc.c 1042 1049 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>pci_enable_device()>pcibios_enable_device()]

```
int pcibios_enable_device(struct pci_dev *dev)
{
    int err;

```

```
	if ((err = pcibios_enable_resources(dev)) < 0)
		return err;
	pcibios_enable_irq(dev);
	return 0;
```

==================== arch/i386/kernel/pci-i386.c 306 332 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>pci_enable_device()>pcibios_enable_device()>pcibios_enable_resources()]

```
int pcibios_enable_resources(struct pci_dev *dev)
{
	u16 cmd, old_cmd;
	int idx;
	struct resource *r;

	pci_read_config_word(dev, PCI_COMMAND, &cmd);
	old_cmd = cmd;
	for(idx=0; idx<6; idx++) {
		r = &dev->resource[idx];
		if (!r->start && r->end) {
			printk(KERN_ERR "PCI: Device %s not available because of resource collisions\n", dev->slot_name);
			return -EINVAL;
		}
		if (r->flags & IORESOURCE_IO)
			cmd |= PCI_COMMAND_IO;
		if (r->flags & IORESOURCE_MEM)
			cmd |= PCI_COMMAND_MEMORY;
	}
	if (dev->resource[PCI_ROM_RESOURCE].start)
		cmd |= PCI_COMMAND_MEMORY;
	if (cmd != old_cmd) {
		printk("PCI: Enabling device %s (%04x -> %04x)\n", dev->slot_name, old_cmd, cmd);
		pci_write_config_word(dev, PCI_COMMAND, cmd);
	}
	return 0;
}
```

我们在前面曾说 USB 设备没有向主机发出中断请求的能力，而只能等待，受主机的查询。但是这并不意味着 USB 控制器（在主机中）没有向 CPU 发出中断请求的能力，这是完全不同的两回事，不能混淆。事实上，USB 控制器是有能力向 CPU 发出中断请求的，所以要接通它的中断请求线（arch/i386/kernel/pci-irq.c）。

==================== arch/i386/kernel/pci-irq.c 613 628 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>pci_enable_device()>pcibios_enable_device()>pcibios_enable_irq()]

```
void pcibios_enable_irq(struct pci_dev *dev)
{
	u8 pin;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &pin);
	if (pin && !pcibios_lookup_irq(dev, 1) && !dev->irq) {
		char *msg;
		if (io_apic_assign_pci_irqs)
			msg = " Probably buggy MP table.";
		else if (pci_probe & PCI_BIOS_IRQ_SCAN)
			msg = "";
		else
			msg = " Please try using pci=biosirq.";
		printk(KERN_WARNING "PCI: No IRQ known for interrupt pin %c of device %s.%s\n",
			  'A' + pin - 1, dev->slot_name, msg);
	}
}
```

USB 控制器本身带有微处理器，在 USB 总线上发送 / 接收的信息都由 USB 控制器通过 DMA 直接从内存读 / 写，主机的 CPU 只要提供缓冲区指针就可以了。而且，CPU 也不需要逐次地为 USB 总线上的操作提供缓冲区指针，而只要把缓冲区指针记录在相应的交互请求，或曰交互描述块中就可以了。USB 控制器自会顺着交互请求队列逐个地完成对这些缓冲区的操作。类似的 DMA 操作称为“智能化 DMA”。其实，USB 总线控制器的 DMA 操作甚至比一般的智能化 DMA 还要复杂，因为 CPU 为之准备的并不只是一个缓冲区从列，而是许多交互请求队列，称为一个“调度”。

PCI 设备（的接口）要进行 DMA 操作就得具有竞争成为“总线主”的能力。另一方面，PCI 设备的 DMA 功能还要服从 CPU 的统一管理，在 PCI 配置寄存器组的命令寄存器中有一个控制位 PCI_COMMAND_MASTER，就是用来打开或关闭具体 PCI 设备竞争成为总线主的能力。在完成 PCI 总线的初始化时，所有 PCI 设备的 DMA 功能都是关闭的，所以这里要通过 pci_set_master()启用 USB 控制器竞争成为总线主的能力（drivers/pci/pci.c）。

==================== drivers/pci/pci.c 508 520 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>pci_set_master()]

```
void
pci_set_master(struct pci_dev *dev)
{
	u16 cmd;

	pci_read_config_word(dev, PCI_COMMAND, &cmd);
	if (! (cmd & PCI_COMMAND_MASTER)) {
		DBG("PCI: Enabling bus mastering for device %s\n", dev->slot_name);
		cmd |= PCI_COMMAND_MASTER;
		pci_write_config_word(dev, PCI_COMMAND, cmd);
	}
	pcibios_set_master(dev);
}
```

再回到 uhci_pci_probe()的代码中，USB 控制器作为 PCI 设备在 PCI 总线层次上的初始化已经完成了，下面就是 USB 总线控制器本身的初始化，即 USB 总线的初始化了。这是由 setup_uhci()完成的，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 2327 2374 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()]

```
/*
 * If we've successfully found a UHCI, now is the time to increment the
 * module usage count, and return success..
 */
static int setup_uhci(struct pci_dev *dev, int irq, unsigned int io_addr, unsigned int io_size)
{
	int retval;
	struct uhci *uhci;
	char buf[8], *bufp = buf;

#ifndef __sparc__
	sprintf(buf, "%d", irq);
#else
	bufp = __irq_itoa(irq);
#endif
	printk(KERN_INFO __FILE__ ": USB UHCI at I/O 0x%x, IRQ %s\n",
		io_addr, bufp);

	uhci = alloc_uhci(io_addr, io_size);
	if (!uhci)
		return -ENOMEM;
	dev->driver_data = uhci;

	request_region(uhci->io_addr, io_size, "usb-uhci");

	reset_hc(uhci);

	usb_register_bus(uhci->bus);
	start_hc(uhci);

	retval = -EBUSY;
	if (request_irq(irq, uhci_interrupt, SA_SHIRQ, "usb-uhci", uhci) == 0) {
		uhci->irq = irq;

		pci_write_config_word(dev, USBLEGSUP, USBLEGSUP_DEFAULT);

		if (!uhci_start_root_hub(uhci))
			return 0;
	}

	/* Couldn't allocate IRQ if we got here */

	reset_hc(uhci);
	release_region(uhci->io_addr, uhci->io_size);
	release_uhci(uhci);

	return retval;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

参数 dev 指向 USB 总线控制器的 pci_dev 数据结构，irq 为该 PCI 设备所连接的中断请求输入线号，io_addr 则为其 I/O 地址空间的起始地址，区间的大小为 io_size。

每个 USB 控制器控制着一条 USB 总线，需要有一个数据结构作为代表。对于遵循 UHCI 界面的控制器，那就是 uhci 结构，定义于 drivers/usb/uhci.h：

==================== drivers/usb/uhci.h 308 335 ====================

```
struct uhci {
        /* Grabbed from PCI */
        int irq;
        unsigned int io_addr;
        unsigned int io_size;

        struct list_head uhci_list;

        struct usb_bus *bus;

        struct uhci_td skeltd[UHCI_NUM_SKELTD];    /* Skeleton TD's */
        struct uhci_qh skelqh[UHCI_NUM_SKELQH];    /* Skeleton QH's */

        spinlock_t framelist_lock;
        struct uhci_framelist *fl;        /* Frame list */
        int fsbr;               /* Full speed bandwidth reclamation */

        spinlock_t qh_remove_lock;
        struct list_head qh_remove_list;

        spinlock_t urb_remove_lock;
        struct list_head urb_remove_list;

        struct s_nested_lock urblist_lock;
        struct list_head urb_list;

        struct virt_root_hub rh;    /* private data of the virtual root hub */
};
```

数据结构中有几个特别重要的成分。首先是指针分 fl，指向一个 uhci_framelist 数据结构，这就是具体 USB 总线的“框架表”，这种数据结构也定义于 drivers/usb/uhci.h：

==================== drivers/usb/uhci.h 100 102 ====================

```
struct uhci_framelist {
        __u32 frame[UHCI_NUMFRAMES];
} __attribute__((aligned(4096)));
```

USB 总线的框架表实际上是个指针数组，每个指针都指向一个等时交互队列。常数 UHCI_NUMFRAMES 在同一文件中定义为 1024，所以整个数组代表着 1024 个框架。数组的起始地址必须与 4K 字节边界对齐，这样其起始地址的低 12 位就全都是 0。USB 控制器内部有个“框架表基地

PDF-XCHANGE Click to buy NOW! www.docu-track.com

址寄存器”，用来记录这个基地址。同时，USB 控制器内部还有个 10 位的“框架计数器”，这个计数器从 0 开始，每过 1 毫秒（1/1024 秒）就加 1，直至 0x3ff 即 1023，然后又变为 0，如此周而复始。在框架计数的后面添上两位 0，再与框架表的基地址连在一起，就成了指向框架表中某个表项的指针。框架表中的每个表项都指向一个 uhci_td 结构的队列，每个 uhci_td 结构是对一个交互的描述，我们称之为“交互描述块”或“交互请求”。USB 控制器在每个框架中首先就执行这个队列。每个等时交互队列中的最后一个数据结构指向一个（实际上是一截）中断交互队列。中断交互队列与框架之间并不是一一对应的关系，uhci 结构中有个 uhci_td 结构数组 skeltd[]，其中的每个元素都指向一截中断交互请求队列，常数 UHCI_NUM_SKELTD 定义为 10，其中 skeltd[0]是整个队列的终点，而 skeltd[9]实际上不用，所以一共有 8 截这样的中断交互请求队列。这些中断交互请求队列又链接在一起，成为一个总的中断交互请求队列。但是，链接在不同部位上的中断交互请求受到执行的频率是不一样的。当将一个代表着中断交互的 uhci_td 结构链入队列时，可以根据所要求的执行周期选择链入中断交互请求队列的不同部位。而 skeltd[]中的各个元素，则起着链入点的作用，所以这个数组称为中断交互请求队列的“骨架”（skeleton）。这些链入点本身也是 uhci_td 结构，不过是空闲的 uhci_td 结构，USB 控制器在执行时会自动跳过。此外，下面读者将会看到，虽然中断交互请求队列并不与框架一一对应，二者间还是有着某种静态的对应关系。

等时交互和中断交互都是周期性的，在每个框架中二者的流量加在一起不超过 90%。这样，在每个框架中，USB 控制器在执行完这两种交互请求以后总是还有一些时间（至少 10%），可以用来执行控制交互以及成块交互。这两种交互都不是周期性的，其队列与框架没有静态的对虑关系，USB 控制器对这些队列的执行完全是动态的，有时间就执行，没有时间就不执行，时间多就多执行，时间少就少执行。在 uhci 结构中还有个 uhci_qh 结构数组 skelqh[]，数组中的每个元素都是一个队列头，用来维持一个“队列的队列”，或者说传输请求的队列。如前所述，每个传输请求是一个交互请求的队列。所以，这个数组是控制 / 成块传输请求队列的骨架，其大小是 UHCI_NUM_SKELQH，在 drivers/usb/uhci.h 中定义为 4。从逻辑上说，只要有两个链入点就够了，可是实际上 USB 设备有全速和低速之分，还有一个有着特殊的用途，所以共有 4 个。

因此，除还有其它一些成分以外，uhci 数据结构实际上代表着主机 CPU 为一条 USB 总线排好的“日程表”，或者说执行程序，这就称为一个“调度”（schedule）。不言而喻，初始化时要为 USB 控制器分配、建立一个 uhci 数据结构。这是由 alloc_uhci()完成的，其代码在 drivers/usb/uhci.c 中。这个函数的代码较长，我们分段阅读。

==================== drivers/usb/uhci.c 2129 2182 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>alloc_uhci()]

```
/*
 * Allocate a frame list, and then setup the skeleton
 *
 * The hardware doesn't really know any difference
 * in the queues, but the order does matter for the
 * protocols higher up. The order is:
 *
 *  - any isochronous events handled before any
 *    of the queues. We don't do that here, because
 *    we'll create the actual TD entries on demand.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *  - The first queue is the "interrupt queue".
 *  - The second queue is the "control queue", split into low and high speed
 *  - The third queue is "bulk data".
 */
static struct uhci *alloc_uhci(unsigned int io_addr, unsigned int io_size)
{
        int i, port;
        struct uhci *uhci;
        struct usb_bus *bus;

        uhci = kmalloc(sizeof(*uhci), GFP_KERNEL);
        if (!uhci)
                return NULL;

        memset(uhci, 0, sizeof(*uhci));

        uhci->irq = -1;
        uhci->io_addr = io_addr;
        uhci->io_size = io_size;

        spin_lock_init(&uhci->qh_remove_lock);
        INIT_LIST_HEAD(&uhci->qh_remove_list);

        spin_lock_init(&uhci->urb_remove_lock);
        INIT_LIST_HEAD(&uhci->urb_remove_list);

        nested_init(&uhci->urblist_lock);
        INIT_LIST_HEAD(&uhci->urb_list);

        spin_lock_init(&uhci->framelist_lock);

        /* We need exactly one page (per UHCI specs), how convenient */
        /* We assume that one page is atleast 4k (1024 frames * 4 bytes) */
        uhci->fl = (void *)__get_free_page(GFP_KERNEL);
        if (!uhci->fl)
                goto au_free_uhci;

        bus = usb_alloc_bus(&uhci_device_operations);
        if (!bus)
                goto au_free_fl;

        uhci->bus = bus;
        bus->hcpriv = uhci;

```

首先为 uhci 数据结构分配空间，结构中 qh_remove_list、urb_remove_list、urb_list 等队列的作用以后会看到，这里是对这些队列头的初始化。然后就是为框架表分配空间，1024 个指针的大小为 4KB，正好是一个页面。前面讲过，uhci 数据结构代表着一个 USB 控制器，从而也代表着一条 USB 总线；

然而，uhci 数据结构中的成分实际上还不足以全面地反映一条 USB 总线的状态，因此内核中又定义了一种 usb_bus 数据结构。所以，也要为 usb_bus 结构分配空间，并使 uhci 和 usb_bus 两个数据结构通过指针互相指向对方，连结成一个整体。这种数据结构定义于 include/linux/usb.h：

==================== include/linux/usb.h 561 580 ====================

```
struct usb_bus {
    int busnum;             /* Bus number (in order of reg) */

    struct usb_devmap devmap;      /* Device map */
    struct usb_operations *op;     /* Operations (specific to the HC) */
    struct usb_device *root_hub;   /* Root hub */
    struct list_head bus_list;
    void *hcpriv;                  /* Host Controller private data */

    int bandwidth_allocated;    /* on this Host Controller; */
                          /* applies to Int. and Isoc. pipes; */
                          /* measured in microseconds/frame; */
                          /* range is 0..900, where 900 = */
                          /* 90% of a 1-millisecond frame */
    int bandwidth_int_reqs;     /* number of Interrupt requesters */
    int bandwidth_isoc_reqs;    /* number of Isoc. requesters */

    /* usbdevfs inode list */
    struct list_head inodes;
};
```

那么，为什么不把这些内容合在一起，而要分成两个数据结构呢？我们在前面讲过，USB 总线控制器有 UHCI 和 OHCI 两种界面，uhci 数据结构正是对 UHCI 界面的抽象。而 usb_bus 结构，则是对这两种界面的公共部分，即“ USB 总线”层次上的抽象。正因为这样，usb_bus 结构中的指针 hcpriv 是个无类型（void）指针，它可以指向一个 uhci 数据结构，也可以指向一个 ohci 数据结构。

函数 usb_alloc_bus()的作用是为 usb_bus 结构分配空间并进行初始化。由于简单，我们就不列出其代码了。这个数据结构中的指针 op 应该指向一个 usb_operations 数据结构，这里将其设置成指向 uhci_device_operations，分别定义于 include/linux/usb.h 和 drivers/usb/uhci.c：

==================== include/linux/usb.h 550 556 ====================

```
struct usb_operations {
    int (*allocate)(struct usb_device *);
    int (*deallocate)(struct usb_device *);
    int (*get_frame_number) (struct usb_device *usb_dev);
    int (*submit_urb) (struct urb* purb);
    int (*unlink_urb) (struct urb* purb);
};
```

==================== drivers/usb/uhci.c 1615 1621 ====================

```
struct usb_operations uhci_device_operations = {
    uhci_alloc_dev,
    uhci_free_dev,
```

```
    uhci_get_current_frame_number,
    uhci_submit_urb,
    uhci_unlink_urb
};
```

显然，这个数据结构为具体 USB 控制器界面的操作提供了函数指针。

我们继续看 alloc_uhci()的代码。

==================== drivers/usb/uhci.c 2183 2237 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>alloc_uhci()]

```
    /* Initialize the root hub */

    /* UHCI specs says devices must have 2 ports, but goes on to say */
    /*  they may have more but give no way to determine how many they */
    /*  have. However, according to the UHCI spec, Bit 7 is always set */
    /*  to 1. So we try to use this to our advantage */
    for (port = 0; port < (io_size - 0x10) / 2; port++) {
         unsigned int portstatus;

         portstatus = inw(io_addr + 0x10 + (port * 2));
         if (!(portstatus & 0x0080))
              break;
    }
    if (debug)
         info("detected %d ports", port);

    /* This is experimental so anything less than 2 or greater than 8 is */
    /*  something weird and we'll ignore it */
    if (port < 2 || port > 8) {
         info("port count misdetected? forcing to 2 ports");
         port = 2;
    }

    uhci->rh.numports = port;

    /*
     * 9 Interrupt queues; link int2 to int1, int4 to int2, etc
     * then link int1 to control and control to bulk
     */
    for (i = 1; i < 9; i++) {
         struct uhci_td *td = &uhci->skeltd[i];

         uhci_fill_td(td, 0, (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
         td->link = virt_to_bus(&uhci->skeltd[i - 1]);
    }


    uhci_fill_td(&uhci->skel_int1_td, 0, (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
    uhci->skel_int1_td.link = virt_to_bus(&uhci->skel_ls_control_qh) | UHCI_PTR_QH;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    uhci->skel_ls_control_qh.link = virt_to_bus(&uhci->skel_hs_control_qh) | UHCI_PTR_QH;
    uhci->skel_ls_control_qh.element = UHCI_PTR_TERM;

    uhci->skel_hs_control_qh.link = virt_to_bus(&uhci->skel_bulk_qh) | UHCI_PTR_QH;
    uhci->skel_hs_control_qh.element = UHCI_PTR_TERM;

    uhci->skel_bulk_qh.link = virt_to_bus(&uhci->skel_term_qh) | UHCI_PTR_QH;
    uhci->skel_bulk_qh.element = UHCI_PTR_TERM;

    /* This dummy TD is to work around a bug in Intel PIIX controllers */
    uhci_fill_td(&uhci->skel_term_td, 0, (UHCI_NULL_DATA_SIZE << 21) | (0x7f << 8) | USB_PID_IN, 0);
    uhci->skel_term_td.link = UHCI_PTR_TERM;

    uhci->skel_term_qh.link = UHCI_PTR_TERM;
    uhci->skel_term_qh.element = virt_to_bus(&uhci->skel_term_td);
```

USB 总线的根集中器总是与 USB 控制器集成在一起。对于 UHCI 界面的总线控制器，其 I/O 地址区间的前 16 个地址用于总线控制器本身，其余的就用于根集中器。根集中器的每个“端口”（port）占用两个地址。端口的数量则取决于具体的芯片，至少 2 个，最多 8 个。每个端口的状态寄存器中的 bit7 总是 1，所以代码中通过一个循环试读，以确定根集中器中端口的数量。

前面讲过，uhci 结构中的 skeltd[]用于 8 截中断交互队列，是个 uhci_td 数据结构的数组。结构名中的“ td”是“交互描述结构”（transaction descriptor）的意思。这种数据结构定义于 drivers/usb/uhci.h：

==================== drivers/usb/uhci.h 175 190 ====================

```
struct uhci_td {
    /* Hardware fields */
    __u32 link;
    __u32 status;
    __u32 info;
    __u32 buffer;

    /* Software fields */
    unsigned int *frameptr;     /* Frame list pointer */
    struct uhci_td *prevtd, *nexttd; /* Previous and next TD in queue */

    struct usb_device *dev;
    struct urb *urb;        /* URB this TD belongs to */

    struct list_head list;
} __attribute__((aligned(16)));
```

每个 uhci_td 数据结构代表着一个交互请求，就好像是对 USB 控制器的一条指令。结构中前 4 个 32 位长字的作用是由 USB 控制器的硬件结构所确定了的，因而不能改变；不过硬件只认开头这 4 个字段，其余的字段则由软件定义和使用。数据结构的起点必须与 16 字节的边界对齐，这也是 USB 总线控制器的硬件结构所要求的。这 4 个 32 位长字实际上相当于一组寄存器。在本书中，我们把由硬件使用的这一部分称为“交互描述块”，以区别于整个交互描述结构。结构头部的 4 个字段中，link 是指向

PDF-XCHANGE Click to buy NOW! www.docu-track.com

下一个 uhci_td 数据结构的链接指针，buffer 则指向用于发送或接收的缓冲区，二者均为物理地址。此外，info 就好像是命令寄存器，相当于指令中的操作码。内核在 drivers/usb/uhci.c 中提供了一个 inline 函数 uhci_fill_td()，用于设置 uhci_td 数据结构头部除 link 外的三个字段：

==================== drivers/usb/uhci.c 165 171 ====================
```
static void inline uhci_fill_td(struct uhci_td *td, __u32 status,
          __u32 info, __u32 buffer)
{
     td->status = status;
     td->info = info;
     td->buffer = buffer;
}
```

上面 2212 行的 for 循环是对中断交互队列的初始化。除第一个 uhci_td 数据结构 skeltd[0]以外，其余的 8 个 uhci_td 结构都设置成无数据的输入交互；并且都指向其前面的 uhci_td 数据结构；而 status 和 buffer 两个字段则都初始化成 0，表示“空操作”。

至于 skeltd[0]，则（2220～2221 行）初始化成指向低速 USB 设备的控制交互队列，即 uhci->skel_ls_control_qh。这里的 skel_int1_td、skel_ls_control_qh 都是在 drivers/usb/uhci.h 定义的宏定义：

==================== drivers/usb/uhci.h 235 251 ====================
```
#define UHCI_NUM_SKELTD      10
#define skel_int1_td         skeltd[0]
#define skel_int2_td         skeltd[1]
#define skel_int4_td         skeltd[2]
#define skel_int8_td         skeltd[3]
#define skel_int16_td        skeltd[4]
#define skel_int32_td        skeltd[5]
#define skel_int64_td        skeltd[6]
#define skel_int128_td       skeltd[7]
#define skel_int256_td       skeltd[8]
#define skel_term_td         skeltd[9]/* To work around PIIX UHCI bug */

#define UHCI_NUM_SKELQH      4
#define skel_ls_control_qh   skelqh[0]
#define skel_hs_control_qh   skelqh[1]
#define skel_bulk_qh         skelqh[2]
#define skel_term_qh         skelqh[3]
```

这些宏定义实际上说明了各个队列的作用。例如，skel_ls_control_qh 表示 skelqh[0]是低速设备的控制传输队列头。那么，skel_int1_td 又表示什么呢？下面读者就会看到，它表示 skeltd[0]是在每一个时间框架中都会执行一遍的中断交互请求链入点。相应地，skel_int2_td 则表示 skeltd[1]是每两个框架才会执行一遍的中断交互请求链入点，等等。

与 skeltd[]不同，skelqh[]中的元素是 uhci_qh 数据结构，结构名中的“ qh”表示这是个队列头（queue head)。这种数据结构定义于 drivers/usb/uhci.h：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/usb/uhci.h 106 118 ====================

```
struct uhci_qh {
	/* Hardware fields */
	__u32 link;			/* Next queue */
	__u32 element;			/* Queue element pointer */

	/* Software fields */
	/* Can't use list_head since we want a specific order */
	struct usb_device *dev;		/* The owning device */

	struct uhci_qh *prevqh, *nextqh;

	struct list_head remove_list;
} __attribute__((aligned(16)));
```

这个数据结构也与 USB 控制器的硬件有关，其头部的 link 和 element 两个字段的作用是由硬件确定的，因而不能改变，其余的字段则只由软件使用。同样，我们有时把由硬件使用的这一部分称为“队列描述块”，以区别于整个数据结构。字段 link 实际上是指向下一个队列（描述块）的指针；element 也是指针，通常指向本队列中的第一个交互描述块，但是在特殊情况下也可以指向另一个队列描述块。二者均采用物理地址。由于 uhci_td 数据结构和 uhci_qh 数据结构都是与 16 字节边界对齐的，其地址的低 4 位一定是 0，所以 link 和 element 两个字段的低 4 位可以用作标志位。其中最低位为“结束”位 UHCI_PTR_TERM，当队列为空时，将队列头的指针 element 设置成 1（而不是 0），就是因为其“结束 ”位为 1。此外，指针中还有一个标志位 UHCI_PTR_QH，用来告诉 USB 控制器所指向的是队列描述块还是交互描述块。从代码中可以看出，队列头 skel_ls_control_qh 通过其 link 字段指向 skel_hs_control_qh，而 skel_hs_control_qh 则又指向 skel_bulk_qh。所有这三个队列开始时都是空的。除这些以外，skeltd[] 中还有个 skeltd[9]，即 skel_term_td，代码中的注释说这是因为 USB 总线控制器所在的 Intel PIIX 芯片内部有个错误，为了绕过这个错误而设的。还有，在 skelqh[]中还有个 skelqh[3]，即 skel_term_qh，这是为回收框架中尚未用完的时间而设置的，读者以后会看到其作用。

这样，对中断、控制和成块等三种交互请求队列的初始化就完成了。下面是对框架，即 1024 个等时交互请求队列的初始化。我们继续往下看（drivers/usb/uhci.c）。

==================== drivers/usb/uhci.c 2238 2286 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>alloc_uhci()]

```

	/*
	 * Fill the frame list: make all entries point to
	 * the proper interrupt queue.
	 *
	 * This is probably silly, but it's a simple way to
	 * scatter the interrupt queues in a way that gives
	 * us a reasonable dynamic range for irq latencies.
	 */
	for (i = 0; i < 1024; i++) {
		struct uhci_td *irq = &uhci->skel_int1_td;

```

```
2250        if (i & 1) {
2251            irq++;
2252            if (i & 2) {
2253                irq++;
2254                if (i & 4) {
2255                    irq++;
2256                    if (i & 8) {
2257                        irq++;
2258                        if (i & 16) {
2259                            irq++;
2260                            if (i & 32) {
2261                                irq++;
2262                                if (i & 64)
2263                                    irq++;
2264                            }
2265                        }
2266                    }
2267                }
2268            }
2269        }
2270
2271        /* Only place we don't use the frame list routines */
2272        uhci->fl->frame[i] =  virt_to_bus(irq);
2273    }
2274
2275    return uhci;
2276
2277 /*
2278  * error exits:
2279  */
2280 au_free_fl:
2281    free_page((unsigned long)uhci->fl);
2282 au_free_uhci:
2283    kfree(uhci);
2284
2285    return NULL;
2286 }
```

框架表的 1024 个表项代表着 1024 个 1 毫秒框架，每个表项都是一个 uhci_td 结构指针，指向一个等时交互请求队列，而队列中的最后一个 uhci_td 结构则应该指向一个中断交互请求的 uhci_td 结构。然而，一开始时还没有等时交互请求，每个框架的等时交互请求队列都是空的，所以此时框架表的每个表项都应该直接指向中断交互请求队列中的某一个链入点、也就是 skeltd[]中的某个元素。可是，skeltd[]中共有 10 个 uhci_td 数据结构，应该指向哪一个呢？我们看代码，这里的 for 循环是很有意思的。对于框架表中的每一个表项，先都假定指向 skeltd[0]，即 skel_int1_td（2248 行），让中断交互请求 skeltd[0]在每个框架中都会得到一次执行。然后，如果表项的框架号（即数组下标）为奇数（2250 行），则把指针调整为指向 skeltd[1]。这样，skeltd[1]就会每隔一个框架得到一次执行。同时，前面已经看到，skeltd[1]的指针 link 指向 skeltd[0]，所以 skeltd[0]还是在每个框架中都会得到执行，所以，skeltd[0]是 skel_int1_td，

PDF-XCHANGE Click to buy NOW! www.docu-track.com

而 skeltd[1]则是 skel_int2_td。同理，如果框架号的最后两位为 3，则指向 skeltd[2]，使 skeltd[2]每隔 3 个框架就会得到一次执行，所以是 skel_int4_td。由于 skeltd[2]的指针 link 指向 skeltd[1]，所以 skeltd[1]和 skeltd[0]的执行保持不变，余类推。最后，如果框架号的最后 7 位为 127，则指向 skeltd[7]，使其每隔 127 个框架得到一次执行，所以是 skel_int128_td。这样，skeltd[]中的各个 uhci_td 结构就分别会在每 1 毫秒、2 毫秒、4 毫秒、…、128 毫秒中得到一次执行。不过，我们在前面看到，skeltd[]中各个 uhci_td 结构的“操作码”均为 0，即“空操作”，因而实际上不会真的启动一次中断交互，而只是用来起到类似于队列头的作用。假定有个 USB 设备，需要每 16 毫秒对其启动一次中断交互，查询其状态变化，那就可以为之建立一个中断交互请求，并将其插入 skel_int16_td 与 skel_int8_td 之间，即 skeltd[4]和 skeltd[3]之间。理解了这段代码，就明白为什么说 skeltd[]是中断交互请求队列的“骨架”了。读者也许会问，为什么要用 uhci_td 结构来起“类似于队列头”的作用，而不是直接就用队列头，即 skelqh 数据结构呢？这一点后面会专门讲到。

至此，代表着 USB 总线的数据结构就初始化完毕了，我们回到前面 setup_uhci()的代码中（2346 行）。接着，通过 reset_hc()往 USB 总线控制器的命令寄存器中写入一个“全局总清”命令，以保证 USB 总线进入初始状态。这里的“ hc”表示“主控制器”（host controller）。这个函数很简单，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 2087 2096 ====================
[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>reset_hc()]

```
static void reset_hc(struct uhci *uhci)
{
     unsigned int io_addr = uhci->io_addr;

     /* Global reset for 50ms */
     outw(USBCMD_GRESET, io_addr + USBCMD);
     wait_ms(50);
     outw(0, io_addr + USBCMD);
     wait_ms(10);
}
```

创建了代表着 USB 总线的 usb_bus 数据结构以后，还要通过 usb_register_bus()向系统登记，其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 394 416 ====================
[pci_module_init()>pci_register_driver()>pci_announee_device()>uhci_pci_probe()>setup_uhci()>usb_register_bus()]

```
/**
 *   usb_register_bus - registers the USB host controller with the usb core
 *   @bus: pointer to the bus to register
 *
 */
void usb_register_bus(struct usb_bus *bus)
{
     int busnum;

     busnum = find_next_zero_bit(busmap.busmap, USB_MAXBUS, 1);
     if (busnum < USB_MAXBUS) {
```

```
		set_bit(busnum, busmap.busmap);
		bus->busnum = busnum;
	} else
		warn("too many buses");

	/* Add it to the list of buses */
	list_add(&bus->bus_list, &usb_bus_list);

	usbdevfs_add_bus(bus);

	info("new USB bus registered, assigned bus number %d", bus->busnum);
}
```

内核中有个 usb_busmap 数据结构 busmap，里面只有一个成分，也叫 busmap，是用于 USB 总线的位图。此外，还有个队列 usb_bus_list。所谓登记就是从位图中分配一个标志位以及相应的总线号，并将代表着 USB 总线的 usb_bus 数据结构链入队列。

前面在 reset_hc()中对 USB 总线进行了全局总清，接着还要通过 start_hc()进一步设置 USB 控制器，这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 2098 2127 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>usb_register_bus()]

```
static void start_hc(struct uhci *uhci)
{
	unsigned int io_addr = uhci->io_addr;
	int timeout = 1000;

	/*
	 * Reset the HC - this will force us to get a
	 * new notification of any already connected
	 * ports due to the virtual disconnect that it
	 * implies.
	 */
	outw(USBCMD_HCRESET, io_addr + USBCMD);
	while (inw(io_addr + USBCMD) & USBCMD_HCRESET) {
		if (!--timeout) {
			printk(KERN_ERR "uhci: USBCMD_HCRESET timed out!\n");
			break;
		}
	}

	/* Turn on all interrupts */
	outw(USBINTR_TIMEOUT | USBINTR_RESUME | USBINTR_IOC | USBINTR_SP,
		io_addr + USBINTR);

	/* Start at frame 0 */
	outw(0, io_addr + USBFRNUM);
	outl(virt_to_bus(uhci->fl), io_addr + USBFLBASEADD);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    /* Run and mark it configured with a 64-byte max packet */
    outw(USBCMD_RS | USBCMD_CF | USBCMD_MAXP, io_addr + USBCMD);
}
```

这里一方面设置 USB 控制器中的中断控制寄存器，允许其在下列 4 种情况下向主机 CPU 发出中断请求：

(1) USBINTR_TIMEOUT。如果启动一次交互以后目标设备没有回应，从而造成超时。

(2) USBINTR_RESUME。为节约能源，如果一个 USB 设备（包括集中器）在一段时间内没有发生任何交互，就进入类似于“冬眠”的挂起（Suspend）模式，有设备从挂起模式回到正常运行模式时，可以引起一次中断。

(3) USBINTR_IOC。完成了一次交互（Interrupt-On-Completion）。如果某个交互请求表明要在完成以后引起中断，则 USB 控制器会在这个交互所在的框架结束之时发出中断请求。

(4) USBINTR_SP，从目标设备接收到的信包短于预期，称为“短信包”（short-packet）。

另一方面，还将控制器的“框架号寄存器”设成 0，使控制器从 0 号框架开始，并将框架表的基地址设置入“框架基地址寄存器”。最后，将“命令寄存器”中的“启动 / 停止”位 USBCMD_RS 设成 1，USB 控制器就开始运行了。当然，此时的 USB 总线在逻辑上还是空的，因而实际上还不会有交互，也不会有中断清求。此外，虽然总线控制器已开始扫描框架表，根集中器却尚未启用。

USB 控制器的中断服务程序是 uhci_interrupt()，向系统的中断机制登记了中断服务程序（2358 行）以后（以及设置 USB 控制器的 PCI 配置寄存器组中的一个寄存器 USBLEGSUP 以后），便通过 uhci_start_root_hub()启动根集中器的运行（drivers/usb/uhci.c）。

==================== drivers/usb/uhci.c 2307 2325 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()]

```
int uhci_start_root_hub(struct uhci *uhci)
{
    struct usb_device *dev;

    dev = usb_alloc_dev(NULL, uhci->bus);
    if (!dev)
        return -1;

    uhci->bus->root_hub = dev;
    usb_connect(dev);

    if (usb_new_device(dev) != 0) {
        usb_free_dev(dev);

        return -1;
    }

    return 0;
}
```

连接在 USB 总线上的每个设备、包括根集中器、都需要有个 usb_device 数据结构作为代表，这是对 USB 设备的抽象，定义于 include/linux/usb.h 中：

==================== include/linux/usb.h 584 624 ====================

```
struct usb_device {
        int devnum;             /* Device number on USB bus */
        int slow;          /* Slow device? */

        atomic_t refcnt;        /* Reference count */

        unsigned int toggle[2];      /* one bit for each endpoint ([0] = IN, [1] = OUT) */
        unsigned int halted[2];      /* endpoint halts; one bit per endpoint # & direction; */
                                /* [0] = IN, [1] = OUT */
        int epmaxpacketin[16];       /* INput endpoint specific maximums */
        int epmaxpacketout[16];      /* OUTput endpoint specific maximums */

        struct usb_device *parent;
        struct usb_bus *bus;         /* Bus we're part of */

        struct usb_device_descriptor descriptor;/* Descriptor */
        struct usb_config_descriptor *config;/* All of the configs */
        struct usb_config_descriptor *actconfig;/* the active configuration */

        char **rawdescriptors;       /* Raw descriptors for each config */

        int have_langid;        /* whether string_langid is valid yet */
        int string_langid;      /* language ID for strings */

        void *hcpriv;           /* Host Controller private data */

          /* usbdevfs inode list */
        struct list_head inodes;
        struct list_head filelist;

        /*
         * Child devices - these can be either new devices
         * (if this is a hub device), or different instances
         * of this same device.
         *
         * Each instance needs its own set of data structures.
         */

        int maxchild;           /* Number of ports if hub */
        struct usb_device *children[USB_MAXCHILDREN];
};
```

USB 总线上的每个设备都有个动态分配的设备号，由于在一条 USB 总线上的设备不能超过 127 个，所以设备号的范围为 1～127。0 表示尚未分配。如果具体的设备是个集中器，则通过一个指针数组 children[]指向连接在该集中器上的设备（usb_device 结构）。而每个设备（除根集中器之外），则通过指针 parent 指向其所连接的集中器（usb_device 结构），同时又通过指针 bus 指向其所在 USB 总线的 usb_bus 数据结构。这样，最终所有的 usb_device 结构就会连接成一个树状的结构，反映出一条 USB 总线的拓

扑图形。此外，usb_device 结构中有个重要的成分 descriptor，是个 usb_device_descriptor 数据结构，定义于 include/linux/usb.h 中：

==================== include/linux/usb.h 218 234 ====================
```
/* Device descriptor */
struct usb_device_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;
        __u16 bcdUSB;
        __u8  bDeviceClass;
        __u8  bDeviceSubClass;
        __u8  bDeviceProtocol;
        __u8  bMaxPacketSize0;
        __u16 idVendor;
        __u16 idProduct;
        __u16 bcdDevice;
        __u8  iManufacturer;
        __u8  iProduct;
        __u8  iSerialNumber;
        __u8  bNumConfigurations;
} __attribute__ ((packed));
```

这个数据结构的内容是由 USB 设备的硬件提供的，其内容固化在设备的硬件中，可以通过一次控制交互从设备读入。显然，其内容与 PCI 设备配置寄存器组中的一些信息相似。不过，USB 控制器在 PCI 总线上，是 PCI 设备；而 USB 设备则直接或间接地（通过集中器）连接在 USB 控制器上，其本身并非 PCI 设备。至于 usb_device 结构中的其它成分，则读者随着代码的阅读自会明白。

像 USB 总线上的其它集中器一样，根集中器本身也是个 USB 设备。所以要为其分配一个 usb_device 结构，并让相应 usb_bus 结构中的指针 root_hub 指向这个数据结构。

然后，还要通过 usb_connect()为这个设备间（？？？）动态地分配一个 USB 总线设备号（drivers/usb/usb.c）。

==================== drivers/usb/usb.c 1667 1698 ====================
[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_connect()]
```
/*
 * Connect a new USB device. This basically just initializes
 * the USB device information and sets up the topology - it's
 * up to the low-level driver to reset the port and actually
 * do the setup (the upper levels don't know how to do that).
 */
void usb_connect(struct usb_device *dev)
{
        int devnum;
        // FIXME needs locking for SMP!!
        /* why? this is called only from the hub thread,
         * which hopefully doesn't run on multiple CPU's simultaneously 8-)
         */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    dev->descriptor.bMaxPacketSize0 = 8;  /* Start off at 8 bytes  */
#ifndef DEVNUM_ROUND_ROBIN
    devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, 1);
#else    /* round_robin alloc of devnums */
    /* Try to allocate the next devnum beginning at devnum_next. */
    devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, devnum_next);
    if (devnum >= 128)
        devnum = find_next_zero_bit(dev->bus->devmap.devicemap, 128, 1);

    devnum_next = devnum + 1;
    if (devnum_next >= 128)
        devnum_next = 1;
#endif    /* round_robin alloc of devnums */

    if (devnum < 128) {
        set_bit(devnum, dev->bus->devmap.devicemap);
        dev->devnum = devnum;
    }
}
```

接下去，就要与目标设备（这里是根集中器）建立起实际的联系了，这就是对目标设备的“枚举”。它是一个颇为复杂的过程。一般而言，当把一个 USB 设备插入一个 USB 集中器的某个“端口”时，集中器就会检测到设备的接入，从而在下一次受到主机通过中断交互查询时就会向其报告。集中器的端口在没有设备连接时都处于关闭状态，插入设备以后也不会自动打开，必须由主机通过控制交互发出命令予以打开。所以，得到集中器的报告以后，主机中的 USB 驱动程序就会为新插上的设备调度若干个控制交互，并向集中器发出打开这个端口的命令。这样，新插入的设备就出现在 USB 总线上了。要在主控制器与 USB 设备之间通信，USB 设备就必须有个（在总线上）惟一的地址。所有的 USB 设备在一开始时都使用“默认”地址 0，与主机中的 USB 控制器建立起初始通信以后再由主机（当然，实际上是驱动软件）指定一个地址，以后就一直使用这个指定的地址，直到断开与总线的连接为止。这里，读者很自然会产生一个问题：如果连接在总线上的所有 USB 设备在一开始时的地址都是 0，那主机的 USB 控制器在初始化时期怎么能与具体的目标设备通信呢？其实很简单，不管有多少个 USB 设备同时连接到各个集中器上，总线控制器总是打开一个端口就指定一个地址，然后再打开下一个端口，决不会在尚未为已打开的端口所连接的设备指定好地址之前就又打开另一个端口。这样，在同一时间里，USB 总线上最多只会有一个设备在使用地址 0。

枚举过程中主机与设备间的信息交换不仅仅是为设备指定地址，总的来说有下面这一些步骤：

(1) 为设备指定地址。

(2) 从设备读入其 usb_device_descriptor 数据结构。

(3) 从设备读入其所有的“配置”描述结构。

(4) 选择或改变设备的配置。

根设备与总线控制器的连接是固定的，不需要通过另一个集中器的报告，所以直接就可以开始其枚举过程，这是由 usb_new_device()完成的，其代码在 drivers/usb/usb.c 中。我们分段阅读。

==================== drivers/usb/usb.c 2079 2133 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()]

```
/*
 * By the time we get here, the device has gotten a new device ID
 * and is in the default state. We need to identify the thing and
 * get the ball rolling..
 *
 * Returns 0 for success, != 0 for error.
 */
int usb_new_device(struct usb_device *dev)
{
    int err;

    /* USB v1.1 5.5.3 */
    /* We read the first 8 bytes from the device descriptor to get to */
    /*  the bMaxPacketSize0 field. Then we set the maximum packet size */
    /*  for the control pipe, and retrieve the rest */
    dev->epmaxpacketin [0] = 8;
    dev->epmaxpacketout[0] = 8;

    err = usb_set_address(dev);
    if (err < 0) {
        err("USB device not accepting new address=%d (error=%d)",
            dev->devnum, err);
        clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
        dev->devnum = -1;
        return 1;
    }

    wait_ms(10);  /* Let the SET_ADDRESS settle */

    err = usb_get_descriptor(dev, USB_DT_DEVICE, 0, &dev->descriptor, 8);
    if (err < 8) {
        if (err < 0)
            err("USB device not responding, giving up (error=%d)", err);
        else
            err("USB device descriptor short read (expected %i, got %i)", 8, err);
        clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
        dev->devnum = -1;
        return 1;
    }
    dev->epmaxpacketin [0] = dev->descriptor.bMaxPacketSize0;
    dev->epmaxpacketout[0] = dev->descriptor.bMaxPacketSize0;

    err = usb_get_device_descriptor(dev);
    if (err < sizeof(dev->descriptor)) {
        if (err < 0)
            err("unable to get device descriptor (error=%d)", err);
        else
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
2126                err("USB device descriptor short read (expected %i, got %i)",
2127                    sizeof(dev->descriptor), err);
2128
2129            clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
2130            dev->devnum = -1;
2131            return 1;
2132        }
2133
```

USB 设备内部有若干用于与主机通信的“端点”(endpoint)，每个端点都有个端点号，不同类型的传输使用不同的端点。每一个 USB 设备至少要提供用于控制传输的控制端点，其端点号为 0。其它端点则视具体设备而定，例如扫描器就不会有等时传输端点，因为不需要。控制端点是双向的，既可用于输入传输，也可用于输出传输。在每次传输中，通信的一方总是主机中的 USB 主控制器，而另一方就由设备号和端点号惟一地确定，称为一个“管道”(pipe)。这里，首先将把目标设备（在这里是根集中器）的控制端点两个方向上的信包大小都假定为 8，然后就通过 usb_set_address()为目标设备指定地址（drivers/usb/usb.c）。

==================== drivers/usb/usb.c 1708 1712 ====================
[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()
>usb_new_device()>usb_set_address()]

```
1708 int usb_set_address(struct usb_device *dev)
1709 {
1710        return usb_control_msg(dev, usb_snddefctrl(dev), USB_REQ_SET_ADDRESS,
1711            0, dev->devnum, 0, NULL, 0, HZ * GET_TIMEOUT);
1712 }
```

这里的 usb_control_msg()专门用来调度控制传输并等待传输的完成。我们将在后面讲述具体 USB 设备的操作时详细介绍这个函数，现在暂且假定调用这个函数就可以让 USB 控制器把一个控制报文发送给目标设备。如果控制传输顺利完成就返回 0，否则返回一个负的出错代码。调用的参数表明，使用的管道是“snddefctrl”，即默认的地址 0 加上控制端点号 0，交互的方向为输出。此外，发送给设备的命令码是 USB_REQ_SET_ADDRESS，即设置地址；而所设置的地址为 dev->devnum，即目标设备的设备号；没有附加数据；允许等待传输完成的时间是 3 秒，即 HZ*GET_TIMEOUT，这里 HZ 表示一秒，而 GET_TIMEOUT 则定义为 3。

假定目标设备顺利地完成了设置地址的操作，经过一个短暂的延迟（2106 行）以后，就可以使用新的地址启动另一次控制传输，从目标设备读入其设备描述块，即 usb_device_descriptor 数据结构了。这是由 usb_get_descriptor()完成的，其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1714 1729 ====================
[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()
>usb_new_device()>usb_get_descriptor()]

```
1714 int usb_get_descriptor(struct usb_device *dev, unsigned char type, unsigned char index, void *buf, int
size)
1715 {
1716        int i = 5;
1717        int result;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1718
1719     memset(buf,0,size);    // Make sure we parse really received data
1720
1721     while (i--) {
1722          if ((result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
1723               USB_REQ_GET_DESCRIPTOR, USB_DIR_IN,
1724               (type << 8) + index, 0, buf, size, HZ * GET_TIMEOUT)) > 0 ||
1725              result == -EPIPE)
1726               break;   /* retry if the returned length was 0; flaky device */
1727     }
1728     return result;
1729 }
```

这一次，使用的管道为“rcvctrlpipe”，即目标设备的地址加端点号 0，方向为输入。对设备的命令为 USB_REQ_GET_DESCRIPTOR，即读取描述块。可是，设备中不仅有“设备描述块”，还有“接口描述块”、“配置描述块”等等，所以前面在调用时指明为 USB_DT_DEVICE，即设备描述块。有关的几个常数定义于 include/linux/usb.h：

==================== include/linux/usb.h 44 51 ====================

```
44   /*
45    * Descriptor types
46    */
47   #define USB_DT_DEVICE              0x01
48   #define USB_DT_CONFIG              0x02
49   #define USB_DT_STRING              0x03
50   #define USB_DT_INTERFACE           0x04
51   #define USB_DT_ENDPOINT            0x05
```

调用参数还表明，只从设备描述结构中读取 8 字节。完成了传输，将所需的内容读入 dev->descriptor 以后，还要根据来自设备的数据，调整控制交互信包的最大容量（2118～2119 行）。然后，再通过 usb_get_device_descriptor()重读一次目标设备的设备描述结构。其代码在 drivers/usb/usb.c 中。

==================== drivers/usb/usb.c 1746 1757 ====================

[pci_module_init()>pci_register_driver()>pei_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_device_descriptor()]

```
1746 int usb_get_device_descriptor(struct usb_device *dev)
1747 {
1748     int ret = usb_get_descriptor(dev, USB_DT_DEVICE, 0, &dev->descriptor,
1749                     sizeof(dev->descriptor));
1750     if (ret >= 0) {
1751          le16_to_cpus(&dev->descriptor.bcdUSB);
1752          le16_to_cpus(&dev->descriptor.idVendor);
1753          le16_to_cpus(&dev->descriptor.idProduct);
1754          le16_to_cpus(&dev->descriptor.bcdDevice);
1755     }
1756     return ret;
1757 }
```

这一次读入的是整个设备描述结构。那么，为什么先要按 8 个字节先读一次呢？这是因为开始时还不知道对方所支持的信包容量，这个信息在对方的设备描述结构的开头 8 个字节中；然而，8 字节的信包容量是所有设备都支持的，所以先按最低标准先读一次，读入了设备描述结构的开头 8 个字节以后，就知道了对方所支持的最大信包容量，以后就可以按这个容量传输了。此外，从设备读入的 16 位整数都是 “ little ending” 的格式，所以要把它们转换成主机 CPU 所采用的格式。

读入设备描述结构以后，接着还要读入有关设备配置的信息。我们继续往下读 usb_new_device() 的代码（drivers/usb/usb.c）。

==================== drivers/usb/usb.c 2134 2175 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>**usb_new_device()**]

```
	err = usb_get_configuration(dev);
	if (err < 0) {
		err("unable to get device %d configuration (error=%d)",
			dev->devnum, err);
		clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
		dev->devnum = -1;
		usb_free_dev(dev);
		return 1;
	}

	/* we set the default configuration here */
	err = usb_set_configuration(dev, dev->config[0].bConfigurationValue);
	if (err) {
		err("failed to set device %d default configuration (error=%d)",
			dev->devnum, err);
		clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
		dev->devnum = -1;
		return 1;
	}

	dbg("new device strings: Mfr=%d, Product=%d, SerialNumber=%d",
		dev->descriptor.iManufacturer, dev->descriptor.iProduct, dev->descriptor.iSerialNumber);
#ifdef DEBUG
	if (dev->descriptor.iManufacturer)
		usb_show_string(dev, "Manufacturer", dev->descriptor.iManufacturer);
	if (dev->descriptor.iProduct)
		usb_show_string(dev, "Product", dev->descriptor.iProduct);
	if (dev->descriptor.iSerialNumber)
		usb_show_string(dev, "SerialNumber", dev->descriptor.iSerialNumber);
#endif

	/* now that the basic setup is over, add a /proc/bus/usb entry */
	usbdevfs_add_device(dev);

	/* find drivers willing to handle this device */
	usb_find_drivers(dev);
```

```

    /* userspace may load modules and/or configure further */
    call_policy ("add", dev);

    return 0;
}
```

设备的每一种具体的配置都由一个 usb_config_descriptor 数据结构加以描述，定义于 include/linux/usb.h：

==================== include/linux/usb.h 280 295 ====================

```
280 /* Configuration descriptor information.. */
281 struct usb_config_descriptor {
282     __u8  bLength         __attribute__ ((packed));
283     __u8  bDescriptorType    __attribute__ ((packed));
284     __u16 wTotalLength __attribute__ ((packed));
285     __u8  bNumInterfaces     __attribute__ ((packed));
286     __u8  bConfigurationValue __attribute__ ((packed));
287     __u8  iConfiguration     __attribute__ ((packed));
288     __u8  bmAttributes __attribute__ ((packed));
289     __u8  MaxPower        __attribute__ ((packed));
290 
291     struct usb_interface *interface;
292 
293     unsigned char *extra;   /* Extra descriptors */
294     int extralen;
295 };
```

结构中指针 interface 以前的部分（282～289 行）与直接从设备读取的“配置描述块”格式相同。函数 usb_get_configuration()的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1926 2016 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_configuration()]

```
int usb_get_configuration(struct usb_device *dev)
{
    int result;
    unsigned int cfgno, length;
    unsigned char buffer[8];
    unsigned char *bigbuffer;
    struct usb_config_descriptor *desc =
        (struct usb_config_descriptor *)buffer;

    if (dev->descriptor.bNumConfigurations > USB_MAXCONFIG) {
        warn("too many configurations");
        return -EINVAL;
    }

    if (dev->descriptor.bNumConfigurations < 1) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        warn("not enough configurations");
        return -EINVAL;
    }

    dev->config = (struct usb_config_descriptor *)
        kmalloc(dev->descriptor.bNumConfigurations *
        sizeof(struct usb_config_descriptor), GFP_KERNEL);
    if (!dev->config) {
        err("out of memory");
        return -ENOMEM;
    }
    memset(dev->config, 0, dev->descriptor.bNumConfigurations *
        sizeof(struct usb_config_descriptor));

    dev->rawdescriptors = (char **)kmalloc(sizeof(char *) *
        dev->descriptor.bNumConfigurations, GFP_KERNEL);
    if (!dev->rawdescriptors) {
        err("out of memory");
        return -ENOMEM;
    }

    for (cfgno = 0; cfgno < dev->descriptor.bNumConfigurations; cfgno++) {
        /* We grab the first 8 bytes so we know how long the whole */
        /*  configuration is */
        result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, buffer, 8);
        if (result < 8) {
            if (result < 0)
                err("unable to get descriptor");
            else {
                err("config descriptor too short (expected %i, got %i)", 8, result);
                result = -EINVAL;
            }
            goto err;
        }

        /* Get the full buffer */
        length = le16_to_cpu(desc->wTotalLength);

        bigbuffer = kmalloc(length, GFP_KERNEL);
        if (!bigbuffer) {
            err("unable to allocate memory for configuration descriptors");
            result = -ENOMEM;
            goto err;
        }

        /* Now that we know the length, get the whole thing */
        result = usb_get_descriptor(dev, USB_DT_CONFIG, cfgno, bigbuffer, length);
        if (result < 0) {
            err("couldn't get all of config descriptors");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1990                kfree(bigbuffer);
1991                goto err;
1992            }
1993
1994            if (result < length) {
1995                err("config descriptor too short (expected %i, got %i)", length, result);
1996                result = -EINVAL;
1997                kfree(bigbuffer);
1998                goto err;
1999            }
2000
2001            dev->rawdescriptors[cfgno] = bigbuffer;
2002
2003            result = usb_parse_configuration(dev, &dev->config[cfgno], bigbuffer);
2004            if (result > 0)
2005                dbg("descriptor data left");
2006            else if (result < 0) {
2007                result = -EINVAL;
2008                goto err;
2009            }
2010        }
2011
2012        return 0;
2013 err:
2014        dev->descriptor.bNumConfigurations = cfgno;
2015        return result;
2016 }
```

每个 USB 设备至少有一个配置描述块。设备描述块中的字段 bNumConfigurations 说明了本设备有几个配置描述块，但最多不能超过 USB_MAXCONFIG，即 8 个。根据这个字段的数值，可以为目标设备分配用于相应描述结构的空间，并让目标设备的 usb_device 数据结构通过其指针 config 指向这块空间。此外，usb_device 数据结构中还有个指针 rawdescriptors，应该指向一个指针数组，该数组中的每一个元素都指向一个从设备读入的配置描述块，所以其大小也取决于配置描述块的个数。做好了这些准备以后，就通过一个 for 循环（1962 行）从设备依次读入各个配置描述块。配置描述块的读入还是由 usb_get_descriptor()完成，但是从代码中可以看出现在要读入的是 USB_DT_CONFIG。每次先读入 8 个字节，读入的信息暂时存放在 buffer 中。根据读入信息中的 wTotalLength 可以知道实际的大小，然后再分配足够大的空间，再调用一次 usb_get_descriptor()，把整个描述块读进来，并将其起始地址放在 rawdescriptors 所指的指针数组中。

每个配置描述块是作为一个整体读入缓冲区的。配置描述块中包含着若干个“接口描述块”，而每个接口描述块中又可以包含若干个“端点描述块”，各种次层描述块的数量则因具体的配置而异，所以配置描述块的大小并非常数。同时，这些描述块中又可以包含一些由具体设备的制造商或行业协会自行定义的次层描述块。所以，每读入一个配置描述块以后，都要通过 usb_parse_configuration()加以分析辩认，从配置描述块中分解出各个次层描述块，并为这些描述块建立起相应的数据结构，以形成对目标设备各个层次的描述。其代码在 drivers/usb/usb.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== drivers/usb/usb.c 1399 1488 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_configuration()>usb_parse_configuration()]

```
int usb_parse_configuration(struct usb_device *dev, struct usb_config_descriptor *config, char
*buffer)
{
     int i, retval, size;
     struct usb_descriptor_header *header;

     memcpy(config, buffer, USB_DT_CONFIG_SIZE);
     le16_to_cpus(&config->wTotalLength);
     size = config->wTotalLength;

     if (config->bNumInterfaces > USB_MAXINTERFACES) {
          warn("too many interfaces");
          return -1;
     }

     config->interface = (struct usb_interface *)
          kmalloc(config->bNumInterfaces *
          sizeof(struct usb_interface), GFP_KERNEL);
     dbg("kmalloc IF %p, numif %i", config->interface, config->bNumInterfaces);
     if (!config->interface) {
          err("out of memory");
          return -1;
     }

     memset(config->interface, 0,
           config->bNumInterfaces * sizeof(struct usb_interface));

     buffer += config->bLength;
     size -= config->bLength;

     for (i = 0; i < config->bNumInterfaces; i++) {
          int numskipped, len;
          char *begin;

          /* Skip over the rest of the Class Specific or Vendor */
          /*  Specific descriptors */
          begin = buffer;
          numskipped = 0;
          while (size >= sizeof(struct usb_descriptor_header)) {
               header = (struct usb_descriptor_header *)buffer;

               if ((header->bLength > size) || (header->bLength < 2)) {
                    err("invalid descriptor length of %d", header->bLength);
                    return -1;
               }

```

```
            /* If we find another descriptor which is at or below */
            /*  us in the descriptor heirarchy then we're done  */
            if ((header->bDescriptorType == USB_DT_ENDPOINT) ||
                (header->bDescriptorType == USB_DT_INTERFACE) ||
                (header->bDescriptorType == USB_DT_CONFIG) ||
                (header->bDescriptorType == USB_DT_DEVICE))
                 break;

            dbg("skipping descriptor 0x%X", header->bDescriptorType);
            numskipped++;

            buffer += header->bLength;
            size -= header->bLength;
        }
        if (numskipped)
            dbg("skipped %d class/vendor specific endpoint descriptors", numskipped);

        /* Copy any unknown descriptors into a storage area for */
        /*  drivers to later parse */
        len = (int)(buffer - begin);
        if (!len) {
            config->extra = NULL;
            config->extralen = 0;
        } else {
            config->extra = kmalloc(len, GFP_KERNEL);
            if (!config->extra) {
                err("couldn't allocate memory for config extra descriptors");
                config->extralen = 0;
                return -1;
            }

            memcpy(config->extra, begin, len);
            config->extralen = len;
        }

        retval = usb_parse_interface(dev, config->interface + i, buffer, size);
        if (retval < 0)
            return retval;

        buffer += retval;
        size -= retval;
    }

    return size;
}
```

参数 config 指向一个 usb_config_descriptor 数据结构，而 buffer 则指向刚从设备读入的“原始”配置描述块缓冲区。首先将缓冲区的前 9 个字节复制到 usb_config_descriptor 结构中，这里的

USB_DT_CONFIG_SIZE 定义为 9。配置描述块中的 bNumInterfaces 说明在本配置中设备中有几个"接口"。所谓一个接口，就是指 USB 设备中的一项特定的功能，也就是逻辑设备。我们可以在逻辑上把每个具体的"接口"看成一组端点的集合，就好像把传统的外设接口看成一组寄存器或存储区间的集合一样。但是，即使是对于同一个接口，即同一个逻辑设备，也还是可以有几种不同的"设置"可供选用，在某种意义上可以看作是同一类逻辑设备的不同实现。这就好像同样是录音机却仍可以组合成不同的性能，将各个按钮用于不同的控制对像。当然，在任何特定的时刻，只能选用其中一种。所以，在内存中为每一个逻辑设备建立起一个 usb_interface 数据结构。这种数据结构定义于 include/linux/usb.h：

==================== include/linux/usb.h 269 278 ====================

```
struct usb_interface {
	struct usb_interface_descriptor *altsetting;

	int act_altsetting;		/* active alternate setting */
	int num_altsetting;		/* number of alternate settings */
	int max_altsetting;		/* total memory allocated */

	struct usb_driver *driver;	/* driver */
	void *private_data;
};
```

结构中的指针 altsetting 指向一个 usb_interface_descriptor 数据结构的数组，数组中的每个元素都是来自设备的"接口描述块"，代表着该接口的一个具体"设置"，而 act_altsetting 则说明当前选用的是其中的哪一个。这个数据结构是在 include/linux/usb.h 定义的：

==================== include/linux/usb.h 251 267 ====================

```
/* Interface descriptor */
struct usb_interface_descriptor {
	__u8  bLength		__attribute__ ((packed));
	__u8  bDescriptorType	__attribute__ ((packed));
	__u8  bInterfaceNumber	__attribute__ ((packed));
	__u8  bAlternateSetting	__attribute__ ((packed));
	__u8  bNumEndpoints	__attribute__ ((packed));
	__u8  bInterfaceClass	__attribute__ ((packed));
	__u8  bInterfaceSubClass __attribute__ ((packed));
	__u8  bInterfaceProtocol __attribute__ ((packed));
	__u8  iInterface	__attribute__ ((packed));

	struct usb_endpoint_descriptor *endpoint;

	unsigned char *extra;   /* Extra descriptors */
	int extralen;
};
```

指针 endpoint 以前的字段均来自目标设备。注意这个结构中的 bInterfaceNumber 和 bAlternateSetting 两个字段。当一个接口有几个不同的"设置"可供选用时，设备就会提供相应数量的接口描述块，这

PDF-XChange Click to buy NOW! www.docu-track.com
PDF-XChange Click to buy NOW! www.docu-track.com

些描述块都有相同的“接口号”，但是“设置号”却各不相同，其中第一个接口描述块的“设置号”为0。所以，在一连串的接口描述块中，每个“设置号”为0的描述块都标志着一个接口的开始。一个接口至少包含一个设置。在配置描述块头部中说明了本配置有几个接口，但是这并不说明有几个接口描述块，所以才需要“ parse”，即分析辩认。

知道了目标设备中有几个接口，就可以为这些接口的 usb_interface 结构分配足够的空间（1413 行），然后通过一个 for 循环（1428 行），依次从已经读入缓冲区的信息中找到属于各个接口的描述块。找到了一个接口描述的开头以后，就通过 usb_parse_interface()进一步分析和辩认出这个接口的所有描述块，其代码也在 drivers/usb/usb.c 中。这个函数比较长，我们得要分段阅读。

==================== drivers/usb/usb.c 1247 1347 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_configuration()>usb_parse_configuration()>usb_parse_interface()]

```
static int usb_parse_interface(struct usb_device *dev, struct usb_interface *interface, unsigned char
*buffer, int size)
{
    int i, len, numskipped, retval, parsed = 0;
    struct usb_descriptor_header *header;
    struct usb_interface_descriptor *ifp;
    unsigned char *begin;

    interface->act_altsetting = 0;
    interface->num_altsetting = 0;
    interface->max_altsetting = USB_ALTSETTINGALLOC;

    interface->altsetting = kmalloc(sizeof(struct usb_interface_descriptor) *
interface->max_altsetting, GFP_KERNEL);

    if (!interface->altsetting) {
        err("couldn't kmalloc interface->altsetting");
        return -1;
    }

    while (size > 0) {
        if (interface->num_altsetting >= interface->max_altsetting) {
            void *ptr;
            int oldmas;

            oldmas = interface->max_altsetting;
            interface->max_altsetting += USB_ALTSETTINGALLOC;
            if (interface->max_altsetting > USB_MAXALTSETTING) {
                warn("too many alternate settings (max %d)",
                    USB_MAXALTSETTING);
                return -1;
            }

            ptr = interface->altsetting;
            interface->altsetting = kmalloc(sizeof(struct usb_interface_descriptor) *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
interface->max_altsetting, GFP_KERNEL);
                if (!interface->altsetting) {
                        err("couldn't kmalloc interface->altsetting");
                        interface->altsetting = ptr;
                        return -1;
                }
                memcpy(interface->altsetting, ptr, sizeof(struct usb_interface_descriptor) * oldmas);

                kfree(ptr);
        }

        ifp = interface->altsetting + interface->num_altsetting;
        interface->num_altsetting++;

        memcpy(ifp, buffer, USB_DT_INTERFACE_SIZE);

        /* Skip over the interface */
        buffer += ifp->bLength;
        parsed += ifp->bLength;
        size -= ifp->bLength;

        begin = buffer;
        numskipped = 0;

        /* Skip over any interface, class or vendor descriptors */
        while (size >= sizeof(struct usb_descriptor_header)) {
                header = (struct usb_descriptor_header *)buffer;

                if (header->bLength < 2) {
                        err("invalid descriptor length of %d", header->bLength);
                        return -1;
                }

                /* If we find another descriptor which is at or below */
                /*  us in the descriptor heirarchy then return */
                if ((header->bDescriptorType == USB_DT_INTERFACE) ||
                    (header->bDescriptorType == USB_DT_ENDPOINT) ||
                    (header->bDescriptorType == USB_DT_CONFIG) ||
                    (header->bDescriptorType == USB_DT_DEVICE))
                        break;

                numskipped++;

                buffer += header->bLength;
                parsed += header->bLength;
                size -= header->bLength;
        }

        if (numskipped)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1328                dbg("skipped %d class/vendor specific interface descriptors", numskipped);
1329
1330           /* Copy any unknown descriptors into a storage area for */
1331           /*  drivers to later parse */
1332           len = (int)(buffer - begin);
1333           if (!len) {
1334                ifp->extra = NULL;
1335                ifp->extralen = 0;
1336           } else {
1337                ifp->extra = kmalloc(len, GFP_KERNEL);
1338
1339                if (!ifp->extra) {
1340                     err("couldn't allocate memory for interface extra descriptors");
1341                     ifp->extralen = 0;
1342                     return -1;
1343                }
1344                memcpy(ifp->extra, begin, len);
1345                ifp->extralen = len;
1346           }
1347
```

在 usb_parse_configuration()中，根据设备在具体配置下接口的个数，为 usb_interface 结构分配了空间，可是还没有为 usb_interface_descriptor 结构分配空间。一般的接口最多可以有 4 个接口描述块，这里的常数 USB_ALTSETTINGALLOC 定义为 4。代码中先按 4 个 usb_interface_descriptor 结构分配空间，然后通过一个 while 循环从前面读自目标设备的缓冲区中寻找和抽取有关的信息。当一个接口中有多于 4 个设置的时候，允许再增加 4 个（1271 行），所以要重新分配一块连续的空间。这里的常数 SB_ALTSETTINGALLOC 也定义为 4，再多就不允许了。

在 while 循环中，首先从缓冲区中复制一个接口描述块，并相应地调整指针 buffer 和 parsed，以及剩余部分的大小。然后就来分析处理这个描述块所提供的信息。

每个接口描述块的开头是两个字节的 usb_descriptor_header 数据结构，说明了描述块的大小与类型，定义于 include/linux/usb.h：

==================== include/linux/usb.h 212 216 ====================

```
/* All standard descriptors have these 2 fields in common */
struct usb_descriptor_header {
     __u8  bLength;
     __u8  bDescriptorType;
} __attribute__ ((packed));
```

如果描述块的类型为 USB_DT_INTERFACE、USB_DT_ENDPOINT、USB_DT_CONFIG、以及 USB_DT_DEVICE 这 4 者之一，就是一个 USB 标准的描述块，甭则就把它跳过，继续往下看下一个描述块（1320～1324 行，以及 1304 行）。被跳过的描述块都是非 USB 标准的，一般是由具体厂商或行业（如数字照相机行业等等）自行定义的。对于这些非 USB 标准的描述块，一方面要通过 dbg()予以报告；另一方面，更重要的是，要分配缓冲区将这些描述块保存下来（1332～1346 行），以备将来由具体设备的驱动程序作进一步的分析，并使当前 usb_interface_descriptor 数据结构中的指针 extra 指向这块

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

缓冲区。

至此，我们已经从原始缓冲区中找到了一个 USB 标准的描述块，下面就要根据这个描述块的类型来决定怎样继续往下分析了。我们继续看 usb_parse_interface()的代码（drivers/usb/usb.c）。

==================== drivers/usb/usb.c 1348 1397 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_configuration()>usb_parse_configuration()>usb_parse_interface()]

```
		/* Did we hit an unexpected descriptor? */
		header = (struct usb_descriptor_header *)buffer;
		if ((size >= sizeof(struct usb_descriptor_header)) &&
		    ((header->bDescriptorType == USB_DT_CONFIG) ||
		     (header->bDescriptorType == USB_DT_DEVICE)))
			return parsed;

		if (ifp->bNumEndpoints > USB_MAXENDPOINTS) {
			warn("too many endpoints");
			return -1;
		}

		ifp->endpoint = (struct usb_endpoint_descriptor *)
			kmalloc(ifp->bNumEndpoints *
			sizeof(struct usb_endpoint_descriptor), GFP_KERNEL);
		if (!ifp->endpoint) {
			err("out of memory");
			return -1;
		}

		memset(ifp->endpoint, 0, ifp->bNumEndpoints *
			sizeof(struct usb_endpoint_descriptor));

		for (i = 0; i < ifp->bNumEndpoints; i++) {
			header = (struct usb_descriptor_header *)buffer;

			if (header->bLength > size) {
				err("ran out of descriptors parsing");
				return -1;
			}

			retval = usb_parse_endpoint(dev, ifp->endpoint + i, buffer, size);
			if (retval < 0)
				return retval;

			buffer += retval;
			parsed += retval;
			size -= retval;
		}

		/* We check to see if it's an alternate to this one */
```

```
1389         ifp = (struct usb_interface_descriptor *)buffer;
1390         if (size < USB_DT_INTERFACE_SIZE ||
1391            ifp->bDescriptorType != USB_DT_INTERFACE ||
1392            !ifp->bAlternateSetting)
1393             return parsed;
1394     }
1395
1396     return parsed;
1397 }
```

如果原始缓冲区中下一个描述块的类型是 USB_DT_CONFIG 或 USB_DT_DEVICE，那就说明对一个新的配置或设备的描述开始了，当前配置中已经再没有更多的接口，所以便返回此时在原始缓冲区中的位置（1353 行），供更高层的函数继续往下分析。否则，如果下一个描述块的类型是 USB_DT_INTERFACE 或 USB_DT_ENDPOINT，便找到了对下一个接口或端点的描述块。在每一个接口描述块的后面是若干端点描述块。

根据已经读入 usb_interface_descriptor 数据结构的信息，可以知道当前“设置”中有几个端点，从而为这些端点的 usb_endpoint_descriptor 数据结构分配空间（1360 行）。这种数据结构是对端点的抽象，定义于 include/linux/usb.h：

==================== include/linux/usb.h 236 249 ====================

```
236  /* Endpoint descriptor */
237  struct usb_endpoint_descriptor {
238      __u8  bLength       __attribute__ ((packed));
239      __u8  bDescriptorType  __attribute__ ((packed));
240      __u8  bEndpointAddress  __attribute__ ((packed));
241      __u8  bmAttributes __attribute__ ((packed));
242      __u16 wMaxPacketSize   __attribute__ ((packed));
243      __u8  bInterval        __attribute__ ((packed));
244      __u8  bRefresh      __attribute__ ((packed));
245      __u8  bSynchAddress     __attribute__ ((packed));
246
247      unsigned char *extra;   /* Extra descriptors */
248      int extralen;
249  };
```

每一个端点都有个固定的端点号，在不同的配置中一个端点可以属于不同的接口，但是其端点号不会改变。另一方面，各个端点的最大信包容量、中断传输的应有周期等等参数也各不相同。

然后，就是通过一个 for 循环依次辩认和抽取各个端点描述块了。同样，每个端点描述块的开头也是由一个 usb_descriptor_header 数据结构说明了描述块的大小。对端点描述块的分析是由 usb_parse_endpoint()完成的，其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1161 1245 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_get_configuration()>usb_parse_configuration()>usb_parse_interface()>usb_parse_endpoint()]

```
1161 static int usb_parse_endpoint(struct usb_device *dev, struct usb_endpoint_descriptor *endpoint,
unsigned char *buffer, int size)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        struct usb_descriptor_header *header;
        unsigned char *begin;
        int parsed = 0, len, numskipped;

        header = (struct usb_descriptor_header *)buffer;

        /* Everything should be fine being passed into here, but we sanity */
        /*  check JIC */
        if (header->bLength > size) {
             err("ran out of descriptors parsing");
             return -1;
        }

        if (header->bDescriptorType != USB_DT_ENDPOINT) {
             warn("unexpected descriptor 0x%X, expecting endpoint descriptor, type 0x%X",
                  endpoint->bDescriptorType, USB_DT_ENDPOINT);
             return parsed;
        }

        if (header->bLength == USB_DT_ENDPOINT_AUDIO_SIZE)
             memcpy(endpoint, buffer, USB_DT_ENDPOINT_AUDIO_SIZE);
        else
             memcpy(endpoint, buffer, USB_DT_ENDPOINT_SIZE);

        le16_to_cpus(&endpoint->wMaxPacketSize);

        buffer += header->bLength;
        size -= header->bLength;
        parsed += header->bLength;

        /* Skip over the rest of the Class Specific or Vendor Specific */
        /*  descriptors */
        begin = buffer;
        numskipped = 0;
        while (size >= sizeof(struct usb_descriptor_header)) {
             header = (struct usb_descriptor_header *)buffer;

             if (header->bLength < 2) {
                  err("invalid descriptor length of %d", header->bLength);
                  return -1;
             }

             /* If we find another descriptor which is at or below us */
             /*  in the descriptor heirarchy then we're done  */
             if ((header->bDescriptorType == USB_DT_ENDPOINT) ||
                 (header->bDescriptorType == USB_DT_INTERFACE) ||
                 (header->bDescriptorType == USB_DT_CONFIG) ||
                 (header->bDescriptorType == USB_DT_DEVICE))
```

```
1211                break;
1212
1213            dbg("skipping descriptor 0x%X",
1214                header->bDescriptorType);
1215            numskipped++;
1216
1217            buffer += header->bLength;
1218            size -= header->bLength;
1219            parsed += header->bLength;
1220        }
1221        if (numskipped)
1222            dbg("skipped %d class/vendor specific endpoint descriptors", numskipped);
1223
1224        /* Copy any unknown descriptors into a storage area for drivers */
1225        /*  to later parse */
1226        len = (int)(buffer - begin);
1227        if (!len) {
1228            endpoint->extra = NULL;
1229            endpoint->extralen = 0;
1230            return parsed;
1231        }
1232
1233        endpoint->extra = kmalloc(len, GFP_KERNEL);
1234
1235        if (!endpoint->extra) {
1236            err("couldn't allocate memory for endpoint extra descriptors");
1237            endpoint->extralen = 0;
1238            return parsed;
1239        }
1240
1241        memcpy(endpoint->extra, begin, len);
1242        endpoint->extralen = len;
1243
1244        return parsed;
1245 }
```

这里所期待的是端点描述块，所以如果描述块的类型不是 USB_DT_ENDPOINT，就立即返回，让高层的函数继续往下分析。一般端点描述块的大小是 7 个字节，但如果是用于音频设备的端点则有 9 个字节。把描述块开头的 7 个或 9 个字节复制到 usb_endpoint_descriptor 数据结构中，并相应推进 buffer 和 parsed，调整还剩下的 size 以后，对一个端点描述块的处理就结束了。但是，在端点描述块这一层上也会有非标准的、由厂家或有关行业自行定义的描述块，所以也要跳过这些描述块，并像前面看到的那样一方面通过 dbg()提出报告，一方面将这些描述块复制下来，留待将来由具体的设备驱动程序加以分析。

回到 usb_parse_interface()的代码中，在穷尽了一个（接口）设置的所有端点描述块以后，就完成了对这个设置的处理。如果原始缓冲区中还有足够的内容，下一个描述块又是个接口描述块，就可以回到 while 循环（1265 行）的开头，继续扫描原始缓冲区中剩下的内容，处理下一个设置了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

处理完一个配置的所有接口以后，便从 usb_parse_configuration()返回到 usb_get_configuration()中继续读入和处理下一个配置，直到完成对设备的所有配置的处理。最后，返回到 usb_new_device()的代码中（2135 行），下一步是通过 usb_set_configuration()将目标设备设置成采用 0 号配置，这是设备备默认的配置。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1884 1910 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_set_configuration()]

```
int usb_set_configuration(struct usb_device *dev, int configuration)
{
	int i, ret;
	struct usb_config_descriptor *cp = NULL;

	for (i=0; i<dev->descriptor.bNumConfigurations; i++) {
		if (dev->config[i].bConfigurationValue == configuration) {
			cp = &dev->config[i];
			break;
		}
	}
	if (!cp) {
		warn("selecting invalid configuration %d", configuration);
		return -EINVAL;
	}

	if ((ret = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
	    USB_REQ_SET_CONFIGURATION, 0, configuration, 0, NULL, 0, HZ * SET_TIMEOUT)) < 0)
		return ret;

	dev->actconfig = cp;
	dev->toggle[0] = 0;
	dev->toggle[1] = 0;
	usb_set_maxpacket(dev);

	return 0;
}
```

至此，作为 PCI 设备，并且作为 USB 控制器的枚举与初始化已经完成了。但是，USB 主控制器必须与总线的根集中器配合才能进行有意义的操作。根集中器是 USB 主控制器的“大门”。而上述过程尚未触及对根集中器的初始化。事实上，根集中器和 USB 总线控制器在物理上总是集成在同一设备中，是同一设备的两个不同“接口”。另一方面，根集中器又是 USB 集中器的一种，在逻辑上和一般的 USB 集中器并无不同，一样适用 USB 集中器的驱动模块。所以对根集中器的初始化只有在安装了 USB 集中器的驱动模块以后才能进行。然而，对 USB 控制器的枚举与初始化跟 USB 集中器驱动模块的安装是两个独立的事件，并无保证何者在先，何者在后。所以，就有了两种可能的情况和对策。

如果 USB 控制器的枚举与初始化在先，则只好推迟 USB 集中器的初始化。到安装 USB 集中器的驱动模块的时候，可以让它来“认领”已经枚举的根集中器，然后对其进行初始化。对于根集中器，这是可能性最大的情况。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

如果 USB 集中器的驱动模块安装在先，则安装时认领不到 USB 集中器，因而无事可做。然后，在完成了 USB 总线控制器的枚举与初始化，并检测到一个具体 USB 集中器的存在时，就设法找到其驱动模块，再通过该驱动模块完成其初始化。根集中器是特殊的 USB 集中器，它的存在是不需要检测的，有 USB 总线控制器就必有根集中器，但是一般的集中器就不同了。实际上，一般 USB 设备的枚举与具体驱动模块的安装更加随机，因为 USB 总线允许“热插入”，随时可以把设备插上拔下。

所以，现在就通过 usb_find_drivers()来找找，看 USB 集中器的驱动模块是否已经安装。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 833 866 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_find_drivers()]

```
/*
 * This entrypoint gets called for each new device.
 *
 * All interfaces are scanned for matching drivers.
 */
static void usb_find_drivers(struct usb_device *dev)
{
     unsigned ifnum;
     unsigned rejected = 0;
     unsigned claimed = 0;

     for (ifnum = 0; ifnum < dev->actconfig->bNumInterfaces; ifnum++) {
          /* if this interface hasn't already been claimed */
          if (!usb_interface_claimed(dev->actconfig->interface + ifnum)) {
               if (usb_find_interface_driver(dev, ifnum))
                    rejected++;
               else
                    claimed++;
          }
     }

     if (rejected)
          dbg("unhandled interfaces on device");

     if (!claimed) {
          warn("USB device %d (vend/prod 0x%x/0x%x) is not claimed by any active driver.",
               dev->devnum,
               dev->descriptor.idVendor,
               dev->descriptor.idProduct);
#ifdef DEBUG
          usb_show_device(dev);
#endif
     }
}
```

对于 USB 主控制器设备的每一个接口（其中之一必定是根集中器），只要尚未被认领，就通过

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

usb_find_interface_driver()扫描已经安装的驱动模块队列并进行比对。其代码在同一文件中：

==================== drivers/usb/usb.c 623 692 ====================

[pci_module_init()>pci_register_driver()>pci_announce_device()>uhci_pci_probe()>setup_uhci()>uhci_start_root_hub()>usb_new_device()>usb_find_drivers()>usb_find_interface_driver()]

```
/*
 * This entrypoint gets called for each new device.
 *
 * We now walk the list of registered USB drivers,
 * looking for one that will accept this interface.
 *
 * "New Style" drivers use a table describing the devices and interfaces
 * they handle.  Those tables are available to user mode tools deciding
 * whether to load driver modules for a new device.
 *
 * The probe return value is changed to be a private pointer.  This way
 * the drivers don't have to dig around in our structures to set the
 * private pointer if they only need one interface.
 *
 * Returns: 0 if a driver accepted the interface, -1 otherwise
 */
static int usb_find_interface_driver(struct usb_device *dev, unsigned ifnum)
{
	struct list_head *tmp;
	struct usb_interface *interface;
	void *private;
	const struct usb_device_id *id;
	struct usb_driver *driver;
	int i;

	if ((!dev) || (ifnum >= dev->actconfig->bNumInterfaces)) {
		err("bad find_interface_driver params");
		return -1;
	}

	interface = dev->actconfig->interface + ifnum;

	if (usb_interface_claimed(interface))
		return -1;

	private = NULL;
	for (tmp = usb_driver_list.next; tmp != &usb_driver_list;) {

		driver = list_entry(tmp, struct usb_driver, driver_list);
		tmp = tmp->next;

		down(&driver->serialize);
		id = driver->id_table;
		/* new style driver? */
```

```
        if (id) {
            for (i = 0; i < interface->num_altsetting; i++) {
                interface->act_altsetting = i;
                id = usb_match_id(dev, interface, id);
                if (id) {
                    private = driver->probe(dev,ifnum,id);
                    if (private != NULL)
                        break;
                }
            }
            /* if driver not bound, leave defaults unchanged */
            if (private == NULL)
                interface->act_altsetting = 0;
        }
        else /* "old style" driver */
            private = driver->probe(dev, ifnum, NULL);

        up(&driver->serialize);
        if (private) {
            usb_driver_claim_interface(driver, interface, private);
            return 0;
        }
    }

    return -1;
}
```

每个已安装的驱动模块都有个“比对表”，说明本模块适用于哪一些或什么样的设备，可以通过 usb_match_id()与具体设备提供的数据比对。我们将在后面结合扫描器的驱动列出这个函数的代码。比对成功，即找到了适用的驱动模块以后，就对目标设备（接口）执行由驱动模块通过函数指针 probe 提供的操作，完成目标设备（接口）的初始化。

在我们这个情景中，假定 USB 集中器的驱动模块尚未安装，所以只好暂缓，等待驱动模块的安装或 USB 集中器初始化的进行。不过，usb_new_device()的代码中还调用了一个函数 call_policy()，这涉及专为“热插入”而设计的一种机制。其大致的作用和过程是让 call_policy()构筑一个命令行“/sbin/hotplug usb”以及必要的环境变量，然后创建起一个内核线程，再让这个线程升级成为一个进程，并执行工具软件/sbin/hotplug 以装入 USB 集中器的驱动模块。限于篇幅，我们这里就从略了。

不管是否出于 call_policy()的间接启动，我们假定 USB 总线的驱动模块或迟或早都会得到安装。安装 USB 总线的驱动模块时，就会执行其初始化程序 usb_init()。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 2243 2250 ====================

```
static int __init usb_init(void)
{
    usb_major_init();
    usbdevfs_init();
    usb_hub_init();

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
2249     return 0;
2250 }
```

就像别的设备驱动程序一样，首先要向系统登记，这是由 usb_major_init()完成的。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 2208 2218 ====================

[usb_init()>usb_major_init()]

```
int usb_major_init(void)
{
    if (devfs_register_chrdev(USB_MAJOR, "usb", &usb_fops)) {
        err("unable to get major %d for usb devices", USB_MAJOR);
        return -EBUSY;
    }

    usb_devfs_handle = devfs_mk_dir(NULL, "usb", NULL);

    return 0;
}
```

对这段代码已经没有什么可说的了。此外，上面 2246 行 usbdevfs_init()的作用只是向 devfs 特殊文件系统登记，读者可参阅"设备文件系统 devfs"一节。所以，usb_init()中关键的操作就是 usb_hub_init()。这个函数的代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 786 808 ====================

[usb_init()>usb_hub_init()]

```
int usb_hub_init(void)
{
    int pid;

    if (usb_register(&hub_driver) < 0) {
        err("Unable to register USB hub driver");
        return -1;
    }

    pid = kernel_thread(usb_hub_thread, NULL,
        CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
    if (pid >= 0) {
        khubd_pid = pid;

        return 0;
    }

    /* Fall through if kernel_thread failed */
    usb_deregister(&hub_driver);
    err("failed to start usb_hub_thread");

    return -1;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

首先通过 usb_register()登记一个 USB 设备驱动模块。每一种 USB 设备都有一个 usb_driver 数据结构，USB 集中器的 usb_driver 数据结构是 hub_driver，定义于 drivers/usb/hub.c：

==================== drivers/usb/hub.c 775 781 ====================

```
static struct usb_driver hub_driver = {
	name:		"hub",
	probe:		hub_probe,
	ioctl:		hub_ioctl,
	disconnect:	hub_disconnect,
	id_table:	hub_id_table,
};
```

我们将在后面以扫描器为实例讲述 USB 设备的初始化时，详细介绍 usb_register()，这里先简略地说明一下。所有的 USB 设备都采用相同的主设备号 USB_MAJOR，而不同类的 USB 设备则由次设备号区分。内核中有个以次设备号为下标的 usb_driver 结构指针数组 usb_minors[]，登记时就根据具体 usb_driver 结构中提供的次设备号将其起始地址填入这个数组中。上面 hub_driver 的定义中没有列出其次设备号，实际上表示 USB 集中器的次设备号为 0。此外，内核中还有个 USB 驱动模块队列 usb_driver_list，登记时把 usb_driver 结构通过其队列头 driver_list 挂入这个队列。然后，还要对已经枚举的所有 USB 设备的 usb_device 结构进行一趟扫描，让新登记的驱动模块“认领”应该由它驱动的设备（或接口），如果找到了就对其执行一次由驱动模块提供的 probe 操作。在我们现在这个情景中，已经枚举的 USB 设备只有一个，那就是根集中器，所以对其调用由 hub_driver 提供的 probe 函数 hub_probe()。其代码在 drivers/usb/hub.c 中（下面的调用路径中跳过了几个函数，后面还会介绍）。

==================== drivers/usb/hub.c 238 320 ====================
[usb_init()>usb_hub_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>hub_probe()]

```
static void *hub_probe(struct usb_device *dev, unsigned int i,
			   const struct usb_device_id *id)

{
	struct usb_interface_descriptor *interface;
	struct usb_endpoint_descriptor *endpoint;
	struct usb_hub *hub;
	unsigned long flags;

	interface = &dev->actconfig->interface[i].altsetting[0];

	/* Some hubs have a subclass of 1, which AFAICT according to the */
	/*  specs is not defined, but it works */
	if ((interface->bInterfaceSubClass != 0) &&
	    (interface->bInterfaceSubClass != 1)) {
		err("invalid subclass (%d) for USB hub device #%d",
			interface->bInterfaceSubClass, dev->devnum);
		return NULL;
	}

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    /* Multiple endpoints? What kind of mutant ninja-hub is this? */
    if (interface->bNumEndpoints != 1) {
        err("invalid bNumEndpoints (%d) for USB hub device #%d",
            interface->bNumEndpoints, dev->devnum);
        return NULL;
    }

    endpoint = &interface->endpoint[0];

    /* Output endpoint? Curiousier and curiousier.. */
    if (!(endpoint->bEndpointAddress & USB_DIR_IN)) {
        err("Device #%d is hub class, but has output endpoint?",
            dev->devnum);
        return NULL;
    }

    /* If it's not an interrupt endpoint, we'd better punt! */
    if ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK) != USB_ENDPOINT_XFER_INT) {
        err("Device #%d is hub class, but has endpoint other than interrupt?",
            dev->devnum);
        return NULL;
    }

    /* We found a hub */
    info("USB hub found");

    hub = kmalloc(sizeof(*hub), GFP_KERNEL);
    if (!hub) {
        err("couldn't kmalloc hub struct");
        return NULL;
    }

    memset(hub, 0, sizeof(*hub));

    INIT_LIST_HEAD(&hub->event_list);
    hub->dev = dev;

    /* Record the new hub's existence */
    spin_lock_irqsave(&hub_event_lock, flags);
    INIT_LIST_HEAD(&hub->hub_list);
    list_add(&hub->hub_list, &hub_list);
    spin_unlock_irqrestore(&hub_event_lock, flags);

    if (usb_hub_configure(hub, endpoint) >= 0)
        return hub;

    err("hub configuration failed for device #%d", dev->devnum);

    /* free hub, but first clean up its list. */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
307      spin_lock_irqsave(&hub_event_lock, flags);
308
309      /* Delete it and then reset it */
310      list_del(&hub->event_list);
311      INIT_LIST_HEAD(&hub->event_list);
312      list_del(&hub->hub_list);
313      INIT_LIST_HEAD(&hub->hub_list);
314
315      spin_unlock_irqrestore(&hub_event_lock, flags);
316
317      kfree(hub);
318
319      return NULL;
320  }
```

参数 dev 指向具体设备（在这里是根集中器）的 usb_device 数据结构，i 表明当前处理的接口号。当设备中有多个接口时，对每个接口都会调用一次相应的 probe 函数。参数 id 则指向一个用于比对、认领的 usb_device_id 数据结构，里面有设备驱动模块所适用的设备类型、制造厂商、产品编号等等信息。由于具体 probe 函数的执行实际上是针对接口（即逻辑设备）的，所以先根据接口号从设备的当前配置中取得指向具体接口描述信息的指针 interface。在调用这个函数之前，已经对设备进行了比对，根据其设备类型、制造厂商等等信息初步认定是个 USB 集中器，这才调用 hub_probe()的。但是，这里还要再进一步作一些检验。USB 集中器的子设备类型号应该是 0 或 1，除默认的控制端点以外应该只有一个中断交互输入端点，251～279 行对这些特征加以检验。如果通过了所谓这些检验，就最后认定了这是个 USB 集中器。虽然 usb_device 数据结构中包含了目标设备作为一般 USB 设备所共有的各种信息，作为具体的 USB 集中器设备还有附加的、反映此种设备特性的信息，以及运行所需的一些字段或成分，所以还要为其分配一个 usb_hub 数据结构。这种数据结构定义于 drivers/usb/hub.h：

==================== drivers/usb/hub.h 93 111 ====================

```
struct usb_hub {
	struct usb_device *dev;

	struct urb *urb;		/* Interrupt polling pipe */

	char buffer[(USB_MAXCHILDREN + 1 + 7) / 8]; /* add 1 bit for hub status change */
				/* and add 7 bits to round up to byte boundary */
	int error;
	int nerrors;

	struct list_head hub_list;

	struct list_head event_list;

	/* Number of ports on the hub */
	int nports;

	struct usb_hub_descriptor *descriptor;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
};
```

结构中有个位图 buffer[]，除其中一位用作总的状态改变标志位以外，每一位都对应着 USB 集中器上的一个端口，反映着该端口的状态，字段 nports 则说明端口的数量。指针 descriptor 指向一个 usb_hub_descriptor 数据结构，定义于同一文件中：

==================== drivers/usb/hub.h 77 89 ====================

```
/* Hub descriptor */
struct usb_hub_descriptor {
        __u8  bLength;
        __u8  bDescriptorType;
        __u8  bNbrPorts;
        __u16 wHubCharacteristics;
        __u8  bPwrOn2PwrGood;
        __u8  bHubContrCurrent;

        /* DeviceRemovable and PortPwrCtrlMask want to be variable-length
           bitmaps that hold max 256 entries, but for now they're ignored */
        __u8  bitmap[0];
} __attribute__ ((packed));
```

显然，这些信息只能来自设备本身。此外，对 USB 集中器的初始化还应该包括为通过周期性的中断传输查询其状态变化作出安排。所以，在对 usb_hub 结构进行一些初步的初始化以后，便通过 usb_hub_configure()进一步完成其初始化。其代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 125 236 ====================

[usb_init()>usb_hub_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>hub_probe()]>usb_hub_configure()]

```
static int usb_hub_configure(struct usb_hub *hub, struct usb_endpoint_descriptor *endpoint)
{
        struct usb_device *dev = hub->dev;
        struct usb_hub_status hubstatus;
        char portstr[USB_MAXCHILDREN + 1];
        unsigned int pipe;
        int i, maxp, ret;

        hub->descriptor = kmalloc(HUB_DESCRIPTOR_MAX_SIZE, GFP_KERNEL);
        if (!hub->descriptor) {
             err("Unable to kmalloc %d bytes for hub descriptor", HUB_DESCRIPTOR_MAX_SIZE);
             return -1;
        }

        /* Request the entire hub descriptor. */
        ret = usb_get_hub_descriptor(dev, hub->descriptor, HUB_DESCRIPTOR_MAX_SIZE);
             /* <hub->descriptor> is large enough for a hub with 127 ports;
              * the hub can/will return fewer bytes here. */
        if (ret < 0) {
             err("Unable to get hub descriptor (err = %d)", ret);
             kfree(hub->descriptor);
```

```
            return -1;
        }

        hub->nports = dev->maxchild = hub->descriptor->bNbrPorts;
        info("%d port%s detected", hub->nports, (hub->nports == 1) ? "" : "s");

        if (hub->descriptor->wHubCharacteristics & HUB_CHAR_COMPOUND)
            dbg("part of a compound device");
        else
            dbg("standalone hub");

        switch (hub->descriptor->wHubCharacteristics & HUB_CHAR_LPSM) {
            case 0x00:
                dbg("ganged power switching");
                break;
            case 0x01:
                dbg("individual port power switching");
                break;
            case 0x02:
            case 0x03:
                dbg("unknown reserved power switching mode");
                break;
        }

        switch (hub->descriptor->wHubCharacteristics & HUB_CHAR_OCPM) {
            case 0x00:
                dbg("global over-current protection");
                break;
            case 0x08:
                dbg("individual port over-current protection");
                break;
            case 0x10:
            case 0x18:
                dbg("no over-current protection");
                        break;
        }

        dbg("power on to power good time: %dms", hub->descriptor->bPwrOn2PwrGood * 2);
        dbg("hub controller current requirement: %dmA", hub->descriptor->bHubContrCurrent);

        for (i = 0; i < dev->maxchild; i++)
            portstr[i] = hub->descriptor->bitmap[((i + 1) / 8)] & (1 << ((i + 1) % 8)) ? 'F' : 'R';
        portstr[dev->maxchild] = 0;

        dbg("port removable status: %s", portstr);

        ret = usb_get_hub_status(dev, &hubstatus);
        if (ret < 0) {
            err("Unable to get hub status (err = %d)", ret);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            kfree(hub->descriptor);
            return -1;
        }

        le16_to_cpus(&hubstatus.wHubStatus);

        dbg("local power source is %s",
            (hubstatus.wHubStatus & HUB_STATUS_LOCAL_POWER) ? "lost (inactive)" : "good");

        dbg("%sover-current condition exists",
            (hubstatus.wHubStatus & HUB_STATUS_OVERCURRENT) ? "" : "no ");

        /* Start the interrupt endpoint */
        pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
        maxp = usb_maxpacket(dev, pipe, usb_pipeout(pipe));

        if (maxp > sizeof(hub->buffer))
            maxp = sizeof(hub->buffer);

        hub->urb = usb_alloc_urb(0);
        if (!hub->urb) {
            err("couldn't allocate interrupt urb");
            kfree(hub->descriptor);
            return -1;
        }

        FILL_INT_URB(hub->urb, dev, pipe, hub->buffer, maxp, hub_irq,
            hub, endpoint->bInterval);
        ret = usb_submit_urb(hub->urb);
        if (ret) {
            err("usb_submit_urb failed (%d)", ret);
            kfree(hub->descriptor);
            return -1;
        }

        /* Wake up khubd */
        wake_up(&khubd_wait);

        usb_hub_power_on(hub);

        return 0;
    }
```

首先为 usb_hub_descriptor 数据结构分配空间，再通过 usb_get_hub_descriptor()从集中器读入所需的信息，即集中器描述块。这个函数的代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 41 46 ====================
[usb_init()>usb_hub_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>hub_probe()>usb_hub_configure()>usb_get_hub_descriptor()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static int usb_get_hub_descriptor(struct usb_device *dev, void *data, int size)
{
     return usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
          USB_REQ_GET_DESCRIPTOR, USB_DIR_IN | USB_RT_HUB,
          USB_DT_HUB << 8, 0, data, size, HZ);
}
```

可见，所需的信息是通过一次控制传输从集中器读入的。读入了集中器描述块以后。就知道了这个集中器有几个端口，也知道了它的其它一些特性。例如，一个集中器可以是一个单纯的 USB 集中器，也可以是一个复合设备中的部分；集中器可以是自带电源的，也可以通过 USB 电缆从主机吸取电流；端口上可以带有过电流保护，也可以不带。不过我们在这里对这些特性不感兴趣。

接着，再通过 usb_get_hub_status() 启动一次控制传输，进一步从集中器读入状态信息（drivers/usb/hub.c），以获取有关其电源供应的当前状况。

==================== drivers/usb/hub.c 66 71 ====================
[usb_init()>usb_hub_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>hub_probe()>usb_hub_configure()>usb_get_hub_status()]

```
static int usb_get_hub_status(struct usb_device *dev, void *data)
{
     return usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
          USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_HUB, 0, 0,
          data, sizeof(struct usb_hub_status), HZ);
}
```

当然，我们对电源也不感必趣。

下面是对中断交互的安排，这才是关键性的。对于 USB 集中器，要为其安排一个或者说“调度”一个周朗性的中断传输（每个中断传输中只有一个交互），使 USB 总线控制器能周期地查询其状态，让它有机会报告各个端口的状态变化。

我们将在后面结合扫描器的驱动介绍对中断传输的调度，这里先作一些简短的说明。

首先要通过 usb_alloc_urb()分配一个“USB 传输请求块”，即 usb 数据结构。再通过宏操作 FILL_INT_URB()设置好这个数据结构，设置的内容包括对方的设备地址与端点号，用来接收信息的缓冲区 hub->buffer，指向具体 usb_hub 数据结构的指针 hub，以及当接收到来自集中器的信息以后需要执行的“中断服务程序”hub_irq()、查询的周期等等。最后通过 usb_submit_urb()提交这个请求，就是根据 urb 数据结构的内容创建一个交互请求，即 uhci_td 数据结构，并根据查询周期将其链入 skeltd[]中某两个元素之间。这样，USB 总线控制器便会周期性地执行这个交互请求，对目标集中器通过中断交互进行查询。

至此，通过 usb_register()对根集中器的枚举与初始化已基本完成，我们随着 CPU 回到 usb_hub_init()的代码中。

阅读过本章“PCI 总线”一节的读者可能已经注意到了，PCI 总线和 USB 总线的枚举过程有明显的不同。PCI 总线的枚举过程是对总线上所有设备的枚举，并且是个递归的过程。而对于 USB 总线，则至今只看到了根集中器的“枚举”，却未见有进一步对连接在集中器上的设备的枚举。虽然 USB 总线的结构也有递归性（多级集中器），但却不见有递归的操作。这实际上反映了二者的一个重要的区别。对于 PCI 总线，虽然在一些特殊的系统中对一些特殊的部件（需要特殊的硬件结构）也有“热插入”

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的要求，但基本的假设是 PCI 设备在系统加电之前就已经静态地连接在总线上。而对 USB 总线却恰好相反，基本的假设是“热插入”，即多数 USB 设备都会在系统加电以后动态地加入或离开系统。在最为一般的情况下，系统初始化时 USB 总线上除根集中器外没有任何设备，而事先有设备连在 USB 总线上反倒是特例。

所以，对于根集中器“后面”的设备的枚举，是一项需要“细水长流”的任务。USB 集中器的驱动模块为此创建了一个内核线程 khubd，专门处理集中器的状态变化。每当将一个 USB 设备插入集中器时，集中器在下一次受到 USB 主控器（通过中断交互）的查询时就会报告状态有了改变，从而 USB 主控制器在当前框架结束时会向 CPU 发出一个中断请求。而 USB 总线的中断服务程序，则会根据在该框架内完成的交互请求，从而 urb 数据结构的内容、调用具体设备的中断服务程序（以后读者会见到）。对于集中器的中断交互，这个中断服务程序是 hub_irq()。其代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 80 110 ====================

```
static void hub_irq(struct urb *urb)
{
	struct usb_hub *hub = (struct usb_hub *)urb->context;
	unsigned long flags;

	/* Cause a hub reset after 10 consecutive errors */
	if (urb->status) {
		if (urb->status == -ENOENT)
			return;

		dbg("nonzero status in irq %d", urb->status);

		if ((++hub->nerrors < 10) || hub->error)
			return;

		hub->error = urb->status;
	}

	hub->nerrors = 0;

	/* Something happened, let khubd figure it out */
	if (waitqueue_active(&khubd_wait)) {
		/* Add the hub to the event queue */
		spin_lock_irqsave(&hub_event_lock, flags);
		if (list_empty(&hub->event_list)) {
			list_add(&hub->event_list, &hub_event_list);
			wake_up(&khubd_wait);
		}
		spin_unlock_irqrestore(&hub_event_lock, flags);
	}
}
```

根据 urb 数据结构的内容可以找到目标集中器的 usb_hub 结构。如果 urb->status 为 0，就表示中断交互正常完成，并且集中器的状态有了变化（如果集中器的状态无变化，则集中器返回 NAK，此时

PDF-XCHANGE Click to buy NOW! www.docu-track.com

USB 控制器不会将该中断交互描述块中的 TD_CTRL_ACTIVE 位清 0，也不会向 CPU 发出中断请求，见后）。只要 khubd 已经在运行，就把目标集中器的 usb_hub 结构通过其队列头 event_list 挂入 khubd 的等待队列 hub_event_list，然后唤醒 khubd。

线程 khubd 是在 usb_hub_init()中（795 行）通过 kernel_thread()创建的，其执行代码 usb_hub_thread()在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 742 765 ====================

```
static int usb_hub_thread(void *__hub)
{
     lock_kernel();

     /*
      * This thread doesn't need any user-level access,
      * so get rid of all our resources
      */

     daemonize();

     /* Setup a nice name */
     strcpy(current->comm, "khubd");

     /* Send me a signal to get me die (for debugging) */
     do {
          usb_hub_events();
          interruptible_sleep_on(&khubd_wait);
     } while (!signal_pending(current));

     dbg("usb_hub_thread exiting");

     up_and_exit(&khubd_exited, 0);
}
```

代码中的 do-while 循环实际上是个无限循环，只要不向这个线程发送信号，就会一直循环下去。在循环体中，先通过 usb_hub_events()检查和处理各个集中器（现在还只有根集中器）的状态变化，然后就睡眠（建议读者思考一下，在循环中为什么处理在先而睡眠在后），直到被唤醒而再次执行 usb_hub_events()。这个函数的代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 622 740 ====================

[usb_hub_thread()>usb_hub_events()]

```
static void usb_hub_events(void)
{
     unsigned long flags;
     struct list_head *tmp;
     struct usb_device *dev;
     struct usb_hub *hub;
     struct usb_hub_status hubsts;
     unsigned short hubstatus, hubchange;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```
	int i, ret;

	/*
	 *  We restart the list everytime to avoid a deadlock with
	 * deleting hubs downstream from this one. This should be
	 * safe since we delete the hub from the event list.
	 * Not the most efficient, but avoids deadlocks.
	 */
	while (1) {
		spin_lock_irqsave(&hub_event_lock, flags);

		if (list_empty(&hub_event_list))
			goto he_unlock;

		/* Grab the next entry from the beginning of the list */
		tmp = hub_event_list.next;

		hub = list_entry(tmp, struct usb_hub, event_list);
		dev = hub->dev;

		list_del(tmp);
		INIT_LIST_HEAD(tmp);

		spin_unlock_irqrestore(&hub_event_lock, flags);

		if (hub->error) {
			dbg("resetting hub %d for error %d", dev->devnum, hub->error);

			if (usb_hub_reset(hub)) {
				err("error resetting hub %d - disconnecting", dev->devnum);
				usb_hub_disconnect(dev);
				continue;
			}

			hub->nerrors = 0;
			hub->error = 0;
		}

		for (i = 0; i < hub->nports; i++) {
			struct usb_port_status portsts;
			unsigned short portstatus, portchange;

			ret = usb_get_port_status(dev, i + 1, &portsts);
			if (ret < 0) {
				err("get_port_status failed (err = %d)", ret);
				continue;
			}

			portstatus = le16_to_cpu(portsts.wPortStatus);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            portchange = le16_to_cpu(portsts.wPortChange);

            if (portchange & USB_PORT_STAT_C_CONNECTION) {
                dbg("port %d connection change", i + 1);

                usb_hub_port_connect_change(dev, i, &portsts);
            } else if (portchange & USB_PORT_STAT_C_ENABLE) {
                dbg("port %d enable change, status %x", i + 1, portstatus);
                usb_clear_port_feature(dev, i + 1, USB_PORT_FEAT_C_ENABLE);

                /*
                 * EM interference sometimes causes bad shielded USB devices to
                 * be shutdown by the hub, this hack enables them again.
                 * Works at least with mouse driver.
                 */
                if (!(portstatus & USB_PORT_STAT_ENABLE) &&
                   (portstatus & USB_PORT_STAT_CONNECTION) && (dev->children[i])) {
                    err("already running port %i disabled by hub (EMI?), re-enabling...",
                        i + 1);
                    usb_hub_port_connect_change(dev, i, &portsts);
                }
            }

            if (portchange & USB_PORT_STAT_C_SUSPEND) {
                dbg("port %d suspend change", i + 1);
                usb_clear_port_feature(dev, i + 1,  USB_PORT_FEAT_C_SUSPEND);
            }

            if (portchange & USB_PORT_STAT_C_OVERCURRENT) {
                err("port %d over-current change", i + 1);
                usb_clear_port_feature(dev, i + 1, USB_PORT_FEAT_C_OVER_CURRENT);
                usb_hub_power_on(hub);
            }

            if (portchange & USB_PORT_STAT_C_RESET) {
                dbg("port %d reset change", i + 1);
                usb_clear_port_feature(dev, i + 1, USB_PORT_FEAT_C_RESET);
            }
        } /* end for i */

        /* deal with hub status changes */
        if (usb_get_hub_status(dev, &hubsts) < 0)
            err("get_hub_status failed");
        else {
            hubstatus = le16_to_cpup(&hubsts.wHubStatus);
            hubchange = le16_to_cpup(&hubsts.wHubChange);
            if (hubchange & HUB_CHANGE_LOCAL_POWER) {
                dbg("hub power change");
                usb_clear_hub_feature(dev, C_HUB_LOCAL_POWER);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                }
                if (hubchange & HUB_CHANGE_OVERCURRENT) {
                    dbg("hub overcurrent change");
                    wait_ms(500); /* Cool down */
                    usb_clear_hub_feature(dev, C_HUB_OVER_CURRENT);
                        usb_hub_power_on(hub);
                }
            }
        } /* end while (1) */

he_unlock:
    spin_unlock_irqrestore(&hub_event_lock, flags);
}
```

通过一个 while 循环，khubd 依次摘下并处理挂在 hub_event_list 中的每一个 usb_hub 数据结构。如果 hub->error 非 0，就表示已经连续出错 10 次以上，所以通过 usb_hub_reset()向相应的集中器发出一个 reset 命令，让它“重新做人”。如果连这也失败，那就要通过 usb_hub_disconnect()断开其连接。

然后，通过一个 for 循环检查和处理集中器的每个端口，因为至此只知道集中器的状态发生了变化，但并不知道具体的情况，那要通过控制传输进一步查询。所以，对于集中器的每个端口，通过 usb_get_port_status()启动一次控制交互读入其状态信息。其代码见 drivers/usb/hub.c：

==================== drivers/usb/hub.c 73 78 ====================

[usb_hub_thread()>usb_hub_events()>usb_get_port_status()]

```
static int usb_get_port_status(struct usb_device *dev, int port, void *data)
{
    return usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
        USB_REQ_GET_STATUS, USB_DIR_IN | USB_RT_PORT, 0, port,
        data, sizeof(struct usb_hub_status), HZ);
}
```

如果具体端口的状态信息表明其设备连接状态发生了变化，那就要作出进一步的处理和反应，这是通过 usb_hub_port_connect_change()完成的。这个函数的代码在 drivers/usb/hub.c 中：

==================== drivers/usb/hub.c 519 620 ====================

[usb_hub_thread()>usb_hub_events()>usb_hub_port_connect_change()]

```
static void usb_hub_port_connect_change(struct usb_device *hub, int port,
                        struct usb_port_status *portsts)
{
    struct usb_device *dev;
    unsigned short portstatus, portchange;
    unsigned int delay = HUB_SHORT_RESET_TIME;
    int i;
    char *portstr, *tempstr;

    portstatus = le16_to_cpu(portsts->wPortStatus);
    portchange = le16_to_cpu(portsts->wPortChange);
    dbg("port %d, portstatus %x, change %x, %s", port + 1, portstatus,
        portchange, portstatus & (1 << USB_PORT_FEAT_LOWSPEED) ? "1.5 Mb/s" : "12 Mb/s");
```

```
    /* Clear the connection change status */
    usb_clear_port_feature(hub, port + 1, USB_PORT_FEAT_C_CONNECTION);

    /* Disconnect any existing devices under this port */
    if (hub->children[port])
        usb_disconnect(&hub->children[port]);

    /* Return now if nothing is connected */
    if (!(portstatus & USB_PORT_STAT_CONNECTION)) {
        if (portstatus & USB_PORT_STAT_ENABLE)
            usb_hub_port_disable(hub, port);

        return;
    }

    down(&usb_address0_sem);

    tempstr = kmalloc(1024, GFP_KERNEL);
    portstr = kmalloc(1024, GFP_KERNEL);

    for (i = 0; i < HUB_PROBE_TRIES; i++) {
        struct usb_device *pdev, *cdev;

        /* Allocate a new device struct */
        dev = usb_alloc_dev(hub, hub->bus);
        if (!dev) {
            err("couldn't allocate usb_device");
            break;
        }

        hub->children[port] = dev;

        /* Reset the device */
        if (usb_hub_port_reset(hub, port, dev, delay)) {
            usb_free_dev(dev);
            break;
        }

        /* Find a new device ID for it */
        usb_connect(dev);

        /* Create a readable topology string */
        cdev = dev;
        pdev = dev->parent;
        if (portstr && tempstr) {
            portstr[0] = 0;
            while (pdev) {
                int port;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

                for (port = 0; port < pdev->maxchild; port++)
                    if (pdev->children[port] == cdev)
                        break;

                strcpy(tempstr, portstr);
                if (!strlen(tempstr))
                    sprintf(portstr, "%d", port + 1);
                else
                    sprintf(portstr, "%d/%s", port + 1, tempstr);

                cdev = pdev;
                pdev = pdev->parent;
            }
            info("USB new device connect on bus%d/%s, assigned device number %d",
                dev->bus->busnum, portstr, dev->devnum);
        } else
            info("USB new device connect on bus%d, assigned device number %d",
                dev->bus->busnum, dev->devnum);

        /* Run it through the hoops (find a driver, etc) */
        if (!usb_new_device(dev))
            goto done;

        /* Free the configuration if there was an error */
        usb_free_dev(dev);

        /* Switch to a long reset time */
        delay = HUB_LONG_RESET_TIME;
    }

    hub->children[port] = NULL;
    usb_hub_port_disable(hub, port);
done:
    up(&usb_address0_sem);
    if (portstr)
        kfree(portstr);
    if (tempstr)
        kfree(tempstr);
}
```

从设备读入的信息中包括两个 16 位状态标志字，一个表示端口当前的状态，一个表示哪些状态发生了变化。在 drivers/usb/hub.h 中定义了一些有关的常数：

==================== drivers/usb/hub.h 40 54 ====================

```
/* wPortStatus bits */
#define USB_PORT_STAT_CONNECTION 0x0001
#define USB_PORT_STAT_ENABLE         0x0002
#define USB_PORT_STAT_SUSPEND        0x0004
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define USB_PORT_STAT_OVERCURRENT     0x0008
#define USB_PORT_STAT_RESET      0x0010
#define USB_PORT_STAT_POWER      0x0100
#define USB_PORT_STAT_LOW_SPEED       0x0200

/* wPortChange bits */
#define USB_PORT_STAT_C_CONNECTION    0x0001
#define USB_PORT_STAT_C_ENABLE        0x0002
#define USB_PORT_STAT_C_SUSPEND       0x0004
#define USB_PORT_STAT_C_OVERCURRENT   0x0008
#define USB_PORT_STAT_C_RESET         0x0010
```

对于连接状态发生了变化的端口，先通过 usb_clear_port_feature()启动一次对目标集中器的控制传输，清除该端口的状态信息（drivers/usb/hub.c）。

==================== drivers/usb/hub.c 54 58 ====================
[usb_hub_thread()>usb_hub_events()>usb_hub_port_connect_change()>usb_clear_port_feature()]

```
static int usb_clear_port_feature(struct usb_device *dev, int port, int feature)
{
     return usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
          USB_REQ_CLEAR_FEATURE, USB_RT_PORT, feature, port, NULL, 0, HZ);
}
```

然后，如果 usb_device 数据结构中的 children[port]非 0，就要调用 usb_disconnect()断开该端口上的连接，这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1619 1665 ====================
[usb_hub_thread()>usb_hub_events()>usb_hub_port_connect_change()>usb_disconnect()]

```
/*
 * Something got disconnected. Get rid of it, and all of its children.
 */
void usb_disconnect(struct usb_device **pdev)
{
     struct usb_device * dev = *pdev;
     int i;

     if (!dev)
          return;

     *pdev = NULL;

     info("USB disconnect on device %d", dev->devnum);

     if (dev->actconfig) {
          for (i = 0; i < dev->actconfig->bNumInterfaces; i++) {
               struct usb_interface *interface = &dev->actconfig->interface[i];
               struct usb_driver *driver = interface->driver;
               if (driver) {
                    down(&driver->serialize);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    driver->disconnect(dev, interface->private_data);
                    up(&driver->serialize);
                    usb_driver_release_interface(driver, interface);
               }
          }
     }

     /* Free up all the children.. */
     for (i = 0; i < USB_MAXCHILDREN; i++) {
          struct usb_device **child = dev->children + i;
          if (*child)
               usb_disconnect(child);
     }

     /* Let policy agent unload modules etc */
     call_policy ("remove", dev);

     /* Free the device number and remove the /proc/bus/usb entry */
     if (dev->devnum > 0) {
          clear_bit(dev->devnum, &dev->bus->devmap.devicemap);
          usbdevfs_remove_device(dev);
     }

     /* Free up the device itself */
     usb_free_dev(dev);
}
```

我们把这个函数留给读者自己阅读。

初看之下这似乎有点奇怪，我们并没有检查这个端口连接状态的变化到底是朝什么方向的变化，怎么能凭 hub->children[port]非 0 就决定断开它的连接呢（537 行）？其实这正是这段代码的精炼之处。如果连接状态的变化是从无到有，那么 hub->children[port]应该是 0，所以实际上不会调用这个函数。万一 hub->children[port]非 0，就说明这是“漏网之鱼”，所以应该将其清除。反过来，如果连接状态的变化是从有到无，那么这就更是我们所需要做的。

进一步，如果端口状态表明无连接（541 行），而端口又处于开通状态（542 行），那就要通过 usb_hub_port_disable()启动另一次控制传输，将目标端口封闭，然后就返回了（545 行）。

如果 CPU 执行到了 548 行，那就说明所发生的变化是从无到有，也就是有个 USB 设备插入了这个端口。这以后的操作，如 usb_alloc_dev()、usb_hub_port_reset()、usb_connect()和 usb_new_device()等，就都是前面已经看到过的了，这就是对新设备的枚举过程。注意这里对 usb_new_device()并非递归调用。如果新的设备又是个集中器，那么 usb_new_device()会通过 usb_find_drivers()找到集中器的驱动模块，并执行其 probe 函数。而集中器驱动模块的 probe 函数又会为新的集中器调度好中断交互，因而 khubd 又会对新的集中器执行 usb_hub_events()。如果不是集中器呢，那就要看具体设备的驱动模块是否已经安装。如果该设备的驱动模块已经安装，那么 usb_new_device()通过 usb_find_drivers()同样会找到其驱动模块，并执行其 probe 函数。反之，如果该设备的驱动模块尚未安装，那就要到安装其驱动模块时再来认领并执行其 probe 函数。下面我们将结合扫描器的驱动说明这个过程。

回到 usb_hub_events()的代码中，对每个端口还要检查和处理其它一些状态变化并作出处理。最后，还要检查整个集中器的电源状态并作出反应（720～735 行）。不过，那些就不是我们在这里所关心的了，有兴趣或需要的读者可自行阅读。

8.9.3 USB 设备的初始化

我们以扫描器为实例来说明一般 USB 设备的初始化过程。

首先，每种 USB 设备都要有个 usb_driver 数据结构，这是对 USB 设备的最高层次上的抽象。以我们在前一节中看到 USB 集中器为例，其 usb_driver 数据结构是 hub_driver。这种数据结构的定义在 include/linux/usb.h 中：

==================== include/linux/usb.h 384 413 ====================

```
struct usb_driver {
    const char *name;

    void *(*probe)(
        struct usb_device *dev,       /* the device */
        unsigned intf,           /* what interface */
        const struct usb_device_id *id    /* from id_table */
        );
    void (*disconnect)(struct usb_device *, void *);

    struct list_head driver_list;

    struct file_operations *fops;
    int minor;

    struct semaphore serialize;

    /* ioctl -- userspace apps can talk to drivers through usbdevfs */
    int (*ioctl)(struct usb_device *dev, unsigned int code, void *buf);

    /* support for "new-style" USB hotplugging
     * binding policy can be driven from user mode too
     */
    const struct usb_device_id *id_table;

    /* suspend before the bus suspends;
     * disconnect or resume when the bus resumes */
    // void (*suspend)(struct usb_device *dev);
    // void (*resume)(struct usb_device *dev);
};
```

扫描器的 usb_driver 结构是 scanner_driver，定义于 drivers/usb/scanner.c：

==================== drivers/usb/scanner.c 953 963 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
953  static struct
954  usb_driver scanner_driver = {
955       name:         "usbscanner",
956       probe:        probe_scanner,
957       disconnect:   disconnect_scanner,
958       fops:         &usb_scanner_fops,
959       minor:        SCN_BASE_MNR,
960       id_table: NULL, /* This would be scanner_device_ids, but we
961                       need to check every USB device, in case
962                       we match a user defined vendor/product ID. */
963  };
```

结构中的成分 probe 和 disconnect 是两个函数指针，分别指向 probe_scanner()和 disconnect_scanner()。顾名思义，前者实际上用于扫描器的初始化，后者则用于断开与扫描器的连接。指针 fops 则指向扫描器设备的 file_operations 数据结构 usb_scanner_fops，也定义于 drivers/usb/scanner.c：

==================== drivers/usb/scanner.c 942 951 ====================

```
942  static struct
943  file_operations usb_scanner_fops = {
944       read:         read_scanner,
945       write:        write_scanner,
946  #ifdef SCN_IOCTL
947       ioctl:        ioctl_scanner,
948  #endif /* SCN_IOCTL */
949       open:         open_scanner,
950       release:  close_scanner,
951  };
```

此外，scanner_driver 中还有个字段 minor，这是具体扫描器的次设备号，其初始值固定为 SCN_BASE_MNR。这个常数定义于 drivers/usb/scanner.h：

==================== drivers/usb/scanner.h 84 85 ====================

```
84   #define SCN_MAX_MNR 16      /* We're allocated 16 minors */
85   #define SCN_BASE_MNR 48     /* USB Scanners start at minor 48 */
```

就是说，扫描器的次设备号从 48 开始，系统中最多可以有 16 台扫描器。

像其它设备一样，作为 USB 设备的扫描器也需要先向系统登记，这就是扫描器驱动模块初始化过程要完成的操作。drivers/usb/scanner.c 中定义了扫描器的初始化函数 usb_scanner_init()。

==================== drivers/usb/scanner.c 971 981 ====================

```
971  int __init
972  usb_scanner_init (void)
973  {
974         if (usb_register(&scanner_driver) < 0)
975                return -1;
976
977       info("USB Scanner support registered.");
978       return 0;
979  }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
module_init(usb_scanner_init);
```

函数 usb_register()的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 85 105 ====================
[usb_scanner_init()>usb_register()]

```
int usb_register(struct usb_driver *new_driver)
{
     if (new_driver->fops != NULL) {
          if (usb_minors[new_driver->minor/16]) {
                err("error registering %s driver", new_driver->name);
               return -EINVAL;
          }
          usb_minors[new_driver->minor/16] = new_driver;
     }

     info("registered new driver %s", new_driver->name);

     init_MUTEX(&new_driver->serialize);

     /* Add it to the list of known drivers */
     list_add_tail(&new_driver->driver_list, &usb_driver_list);

     usb_scan_devices();

     return 0;
}
```

内核中为 USB 设备设立了一个 usb_driver 结构指针数组 usb_minors[]，其大小为 16，每个具体 usb_driver 结构的次设备号除以 16 便决定了它在数组中的位置。每登记一个具体的 usb_driver 结构以后，数组中相应的指针就指向了这个数据结构。由此可见，USB 设备的次设备号实际上分成了两截，其高 4 位起着类似于主设备号的作用，而其低 4 位则是真正的“次设备号”。这样，系统中可以同时存在 16 种不同的 USB 设备，而每种设备（如扫描器）最多可以有 16 台。这当然只是权宜之计，将来随着 devfs 的采用应该会有更好的解决。

为了保证应用进程对 USB 设备操作的互斥性，usb_driver 数据结构中有个内核信号量 serialize，代码中对其进行了初始化。

内核中有两个 USB 层次上的队列。一个是 usb_bus 结构的队列 usb_bus_list，另一个是 usb_driver 结构的队列 usb_driver_list。所有 USB 设备驱动模块的 usb_driver 结构都链接在这个队列中，新驱动模块的 usb_driver 结构就挂在 usb_driver_list 的尾部。

然后，通过 usb_scan_devices()扫描所有 USB 总线上的所有设备，让每个 USB 设备驱动模块都试着来“认领”与其对口的设备。就扫描器的驱动模块而言，就是要让它认领已经枚举的扫描器设备。函数 usb_scan_devices()的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 107 127 ====================
[usb_scanner_init()>usb_register()>usb_scan_devices()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/**
 *   usb_scan_devices - scans all unclaimed USB interfaces
 *
 *   Goes through all unclaimed USB interfaces, and offers them to all
 *   registered USB drivers through the 'probe' function.
 *   This will automatically be called after usb_register is called.
 *   It is called by some of the USB subsystems after one of their subdrivers
 *   are registered.
 */
void usb_scan_devices(void)
{
     struct list_head *tmp;

     tmp = usb_bus_list.next;
     while (tmp != &usb_bus_list) {
          struct usb_bus *bus = list_entry(tmp,struct usb_bus, bus_list);

          tmp = tmp->next;
          usb_check_support(bus->root_hub);
     }
}
```

这个函数扫描队列 usb_bus_list 中的每一个 usb_bus 数据结构，也就是系统中的每一条 USB 总线，通过一个函数 usb_check_support()检查该总线上已经枚举的设备。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 434 458 ====================

[usb_scanner_init()>usb_register()>usb_scan_devices()>usb_check_support()]

```
/*
 * This function is for doing a depth-first search for devices which
 * have support, for dynamic loading of driver modules.
 */
static void usb_check_support(struct usb_device *dev)
{
     int i;

     if (!dev) {
          err("null device being checked!!!");
          return;
     }

     for (i=0; i<USB_MAXCHILDREN; i++)
          if (dev->children[i])
               usb_check_support(dev->children[i]);

     if (!dev->actconfig)
          return;

     /* now we check this device */
     if (dev->devnum > 0)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        for (i = 0; i < dev->actconfig->bNumInterfaces; i++)
            usb_find_interface_driver(dev, i);
}
```

每条 USB 总线的“根”是一个 USB 集中器，那就是根集中器。主机的 USB 控制器一般都与根集中器集成在一起。根集中器也是由 usb_device 数据结构代表的，在 USB 总线初始化时建立起它的数据结构。每个 USB 集中器可以连接若干 USB 设备，其数量取决于具体的集中器，但是最多不超过 USB_MAXCHILDREN，目前这个常数定义为 16。连接在 USB 集中器上的 USB 设备本身又可以是 USB 集中器，所以代码中对下接的集中器递归调用 usb_check_support()，进行深度优先的搜索，一直到不再是集中器，或者尚未插上设备的集中器时才回头。

如果一个 USB 设备已经分配了地址并已成功枚举，也没有被“除名”，其 usb_device 结构中的字段 devnum 就是一个正数，此时结构中的指针 actconfig 指向一个 usb_config_descriptor 数据结构。我们已经在前几个小节中看到这种数据结构的定义。

这个结构中的字段 bNumInterfaces 表示相应的 USB 设备中有着几个逻辑设备，从而有几个“接口”。指针 interface 指向一个 usb_interface 数据结构数组，其定义也已在前几个小节中看到过，数组的大小则取决于 bNumInterfaces。

每一个接口可以包含若干“端点”（endpoint），对于主机来说，每个端点就是一个通信的对象。不管是什么样的 USB 设备，至少要有一个公用的端点用于控制以及枚举的目的，这个端点的“端点号”固定为 0。除此之外，每一个接口还可以有附加的、专有的端点。端点号 0 所代表的通信对象是双向的，既可以收也可以发，而其它端点号所代表的通信对象则都是单向的。这个数据结构中的字段 bNumEndpoints 说明了本接口中有几个端点，而指针 endpoint 则指向一个 usb_endpoint_descriptor 结构数组，每个 usb_endpoint_descriptor 都代表着一个端点，其定义见前一小节。

从代表着具体 USB 总线的 usb_bus 结构到代表着具体设备（包括集中器）的 usb_device 结构，再到 usb_config_descriptor 结构，最后是 usb_interface 结构和 usb_endpoint_descriptor 结构，正好反映了 USB 的各个结构层次。这些数据结构本身以及互相间的联系，都是在 USB 总线初始化时，或者后来当 USB 总线上的设备连接有变化时的枚举阶段建立的。枚举阶段所获取的这些信息反映了当前 USB 总线上各个设备（端点）的客观存在，但是尚未与具体的设备驱动程序挂上钩。所以，在具体设备（类型）的初始化中要在这些数据结构中找到对口的数据结构并加以“认领”（claim）。

函数 usb_find_interface_driver()逐个地检查数组中的每个 usb_interface 数据结构，看看相应逻辑设备的类型是否相符，如果相符就加以初始化。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 623 692 ====================

[usb_scanner_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()]

```
/*
 * This entrypoint gets called for each new device.
 *
 * We now walk the list of registered USB drivers,
 * looking for one that will accept this interface.
 *
 * "New Style" drivers use a table describing the devices and interfaces
 * they handle.  Those tables are available to user mode tools deciding
 * whether to load driver modules for a new device.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *
 * The probe return value is changed to be a private pointer.  This way
 * the drivers don't have to dig around in our structures to set the
 * private pointer if they only need one interface.
 *
 * Returns: 0 if a driver accepted the interface, -1 otherwise
 */
static int usb_find_interface_driver(struct usb_device *dev, unsigned ifnum)
{
    struct list_head *tmp;
    struct usb_interface *interface;
    void *private;
    const struct usb_device_id *id;
    struct usb_driver *driver;
    int i;

    if ((!dev) || (ifnum >= dev->actconfig->bNumInterfaces)) {
        err("bad find_interface_driver params");
        return -1;
    }

    interface = dev->actconfig->interface + ifnum;

    if (usb_interface_claimed(interface))
        return -1;

    private = NULL;
    for (tmp = usb_driver_list.next; tmp != &usb_driver_list;) {

        driver = list_entry(tmp, struct usb_driver, driver_list);
        tmp = tmp->next;

        down(&driver->serialize);
        id = driver->id_table;
        /* new style driver? */
        if (id) {
            for (i = 0; i < interface->num_altsetting; i++) {
                interface->act_altsetting = i;
                id = usb_match_id(dev, interface, id);
                if (id) {
                    private = driver->probe(dev,ifnum,id);
                    if (private != NULL)
                        break;
                }
            }
            /* if driver not bound, leave defaults unchanged */
            if (private == NULL)
                interface->act_altsetting = 0;
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            else /* "old style" driver */
                private = driver->probe(dev, ifnum, NULL);

            up(&driver->serialize);
            if (private) {
                usb_driver_claim_interface(driver, interface, private);
                return 0;
            }
        }

        return -1;
    }
```

这里的参数 dev 指向一个 USB 设备的 usb_device 数据结构，以接口号为下标，就可以在其数组 interface[]中找到具体接口，即逻辑设备的 usb_interface 结构。如果这个结构中的指针 driver 已经指向一个 usb_driver 数据结构，则该接口已被认领。否则，就在 usb_driver_list 队列中扫描已经登记的 usb_driver 数据结构，逐个地通过 usb_match_id()将其 id 表跟设备上的每个接口比对。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 505 621 ====================

[usb_scanner_init()>usb_register()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>usb_match_id()]

```
/* usb_match_id searches an array of usb_device_id's and returns
   the first one that matches the device and interface.

   Parameters:
     "id" is an array of usb_device_id's is terminated by an entry
      containing all zeroes.

      "dev" and "interface" are the device and interface for which
      a match is sought.

   If no match is found or if the "id" pointer is NULL, then
   usb_match_id returns NULL.


   What constitutes a match:

   A zero in any element of a usb_device_id entry is a wildcard
   (i.e., that field always matches).  For there to be a match,
   *every* nonzero element of the usb_device_id must match the
   provided device and interface in.  The comparison is for equality,
   except for one pair of fields: usb_match_id.bcdDevice_{lo,hi} define
   an inclusive range that dev->descriptor.bcdDevice must be in.

   If interface->altsettings does not exist (i.e., there are no
   interfaces defined), then bInterface{Class,SubClass,Protocol}
   only match if they are all zeroes.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```


   What constitutes a good "usb_device_id"?

   The match algorithm is very simple, so that intelligence in
   driver selection must come from smart driver id records.
   Unless you have good reasons to use another selection policy,
   provide match elements only in related groups:

    * device specifiers (vendor and product IDs; and maybe
      a revision range for that product);
    * generic device specs (class/subclass/protocol);
    * interface specs (class/subclass/protocol).

   Within those groups, work from least specific to most specific.
   For example, don't give a product version range without vendor
   and product IDs.

   "driver_info" is not considered by the kernel matching algorithm,
   but you can create a wildcard "matches anything" usb_device_id
   as your driver's "modules.usbmap" entry if you provide only an
   id with a nonzero "driver_info" field.
 */

const struct usb_device_id *
usb_match_id(struct usb_device *dev, struct usb_interface *interface,
         const struct usb_device_id *id)
{
     struct usb_interface_descriptor  *intf = 0;

     /* proc_connectinfo in devio.c may call us with id == NULL. */
     if (id == NULL)
          return NULL;

     /* It is important to check that id->driver_info is nonzero,
        since an entry that is all zeroes except for a nonzero
        id->driver_info is the way to create an entry that
        indicates that the driver want to examine every
        device and interface. */
     for (; id->idVendor || id->bDeviceClass || id->bInterfaceClass ||
            id->driver_info; id++) {

          if ((id->match_flags & USB_DEVICE_ID_MATCH_VENDOR) &&
              id->idVendor != dev->descriptor.idVendor)
               continue;

          if ((id->match_flags & USB_DEVICE_ID_MATCH_PRODUCT) &&
              id->idProduct != dev->descriptor.idProduct)
               continue;
```

```

        /* No need to test id->bcdDevice_lo != 0, since 0 is never
           greater than any unsigned number. */
        if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_LO) &&
            (id->bcdDevice_lo > dev->descriptor.bcdDevice))
             continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_HI) &&
            (id->bcdDevice_hi < dev->descriptor.bcdDevice))
             continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_CLASS) &&
            (id->bDeviceClass != dev->descriptor.bDeviceClass))
             continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_SUBCLASS) &&
            (id->bDeviceSubClass!= dev->descriptor.bDeviceSubClass))
             continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_DEV_PROTOCOL) &&
            (id->bDeviceProtocol != dev->descriptor.bDeviceProtocol))
             continue;

        intf = &interface->altsetting [interface->act_altsetting];

        if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_CLASS) &&
            (id->bInterfaceClass != intf->bInterfaceClass))
             continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_SUBCLASS) &&
            (id->bInterfaceSubClass != intf->bInterfaceSubClass))
            continue;

        if ((id->match_flags & USB_DEVICE_ID_MATCH_INT_PROTOCOL) &&
            (id->bInterfaceProtocol != intf->bInterfaceProtocol))
            continue;

        return id;
    }

    return NULL;
}
```

登记的每个 usb_driver 结构都应该有个 id 表，说明该驱动程序（模块）适用于哪一些设备。所谓 id 表是个 usb_device_id 结构数组，这种数据结构定义于 include/linux/usb.h：

==================== include/linux/usb.h 347 382 ====================

```
struct usb_device_id {
    /* This bitmask is used to determine which of the following fields
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
         * are to be used for matching.
         */
        __u16         match_flags;

        /*
         * vendor/product codes are checked, if vendor is nonzero
         * Range is for device revision (bcdDevice), inclusive;
         * zero values here mean range isn't considered
         */
        __u16         idVendor;
        __u16         idProduct;
        __u16         bcdDevice_lo, bcdDevice_hi;

        /*
         * if device class != 0, these can be match criteria;
         * but only if this bDeviceClass value is nonzero
         */
        __u8     bDeviceClass;
        __u8     bDeviceSubClass;
        __u8     bDeviceProtocol;

        /*
         * if interface class != 0, these can be match criteria;
         * but only if this bInterfaceClass value is nonzero
         */
        __u8     bInterfaceClass;
        __u8     bInterfaceSubClass;
        __u8     bInterfaceProtocol;

        /*
         * for driver's use; not involved in driver matching.
         */
        unsigned long driver_info;
};
```

每个 usb_device_id 结构描述着一个驱动程序可以适用的对象，包括由谁制造、制造商的产品编号、设备的类型、接口的类型等等特征信息。同时，结构中还有个位图 match_flags，说明必须有哪些特征信息都相符才可以把一个设备及接口认定为与给定驱动程序相符。如果 usb_match_id()返回非 0，就说明找到了适用于目标设备的驱动程序。然后，就可以让这个驱动模块的 probe 函数来进行该设备的初始化了。对于扫描器，其 probe 函数为 probe_scanner()，它的代码在 drivers/usb/scanner.c 中。我们分段阅读。

==================== drivers/usb/scanner.c 626 703 ====================

[usb_scanner_init()>usb_registcr()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>probe_scanner()]

```
static void *
probe_scanner(struct usb_device *dev, unsigned int ifnum,
          const struct usb_device_id *id)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
     struct scn_usb_data *scn;
     struct usb_interface_descriptor *interface;
     struct usb_endpoint_descriptor *endpoint;

     int ep_cnt;
     int ix;

     kdev_t scn_minor;

     char valid_device = 0;
     char have_bulk_in, have_bulk_out, have_intr;

     if (vendor != -1 && product != -1) {
          info("probe_scanner: User specified USB scanner -- Vendor:Product - %x:%x", vendor, product);
     }

     dbg("probe_scanner: USB dev address:%p", dev);
     dbg("probe_scanner: ifnum:%u", ifnum);

/*
 * 1. Check Vendor/Product
 * 2. Determine/Assign Bulk Endpoints
 * 3. Determine/Assign Intr Endpoint
 */

/*
 * There doesn't seem to be an imaging class defined in the USB
 * Spec. (yet).  If there is, HP isn't following it and it doesn't
 * look like anybody else is either.  Therefore, we have to test the
 * Vendor and Product ID's to see what we have.  Also, other scanners
 * may be able to use this driver by specifying both vendor and
 * product ID's as options to the scanner module in conf.modules.
 *
 * NOTE: Just because a product is supported here does not mean that
 * applications exist that support the product.  It's in the hopes
 * that this will allow developers a means to produce applications
 * that will support USB products.
 *
 * Until we detect a device which is pleasing, we silently punt.
 */

     for (ix = 0; ix < sizeof (scanner_device_ids) / sizeof (struct usb_device_id); ix++) {
          if ((dev->descriptor.idVendor == scanner_device_ids [ix].idVendor) &&
              (dev->descriptor.idProduct == scanner_device_ids [ix].idProduct)) {
               valid_device = 1;
               break;
              }
     }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
678      if (dev->descriptor.idVendor == vendor &&   /* User specified */
679          dev->descriptor.idProduct == product) { /* User specified */
680            valid_device = 1;
681      }
682
683        if (!valid_device)
684                return NULL;    /* We didn't find anything pleasing */
685
686 /*
687  * After this point we can be a little noisy about what we are trying to
688  *  configure.
689  */
690
691      if (dev->descriptor.bNumConfigurations != 1) {
692           info("probe_scanner: Only one device configuration is supported.");
693           return NULL;
694      }
695
696      if (dev->config[0].bNumInterfaces != 1) {
697           info("probe_scanner: Only one device interface is supported.");
698           return NULL;
699      }
700
701      interface = dev->config[0].interface[ifnum].altsetting;
702      endpoint = interface[ifnum].endpoint;
703
```

首先将目标设备跟扫描器的 id 表 scanner_device_ids 加以核对。然后，使指针 interface 指向目标接口的 usb_interface_descriptor 数据结构。这个结构中的指针 endpoint 指向一个端点描述结构数组，结构中的另一个字段 bNumEndpoints 则记录着这个数组的大小，即本接口的端点数量。数组中的每一个 usb_endpoint_descriptor 结构都代表着目标设备中的一个端点。下面，就要对这些端点进行验证，我们继续往下看 probe_scanner()的代码（drivers/usb/scanner.c）。

==================== drivers/usb/scanner.c 704 769 ====================

[usb_scanner_init()>usb_registcr()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>probe_scanner()]

```
704 /*
705  * Start checking for two bulk endpoints OR two bulk endpoints *and* one
706  * interrupt endpoint. If we have an interrupt endpoint go ahead and
707  * setup the handler. FIXME: This is a future enhancement...
708  */
709
710      dbg("probe_scanner: Number of Endpoints:%d", (int) interface->bNumEndpoints);
711
712      if ((interface->bNumEndpoints != 2) && (interface->bNumEndpoints != 3)) {
713           info("probe_scanner: Only two or three endpoints supported.");
714           return NULL;
715      }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    ep_cnt = have_bulk_in = have_bulk_out = have_intr = 0;

    while (ep_cnt < interface->bNumEndpoints) {

        if (!have_bulk_in && IS_EP_BULK_IN(endpoint[ep_cnt])) {
            ep_cnt++;
            have_bulk_in = ep_cnt;
            dbg("probe_scanner: bulk_in_ep:%d", have_bulk_in);
            continue;
        }

        if (!have_bulk_out && IS_EP_BULK_OUT(endpoint[ep_cnt])) {
            ep_cnt++;
            have_bulk_out = ep_cnt;
            dbg("probe_scanner: bulk_out_ep:%d", have_bulk_out);
            continue;
        }

        if (!have_intr && IS_EP_INTR(endpoint[ep_cnt])) {
            ep_cnt++;
            have_intr = ep_cnt;
            dbg("probe_scanner: intr_ep:%d", have_intr);
            continue;
        }
        info("probe_scanner: Undetected endpoint. Notify the maintainer.");
        return NULL;  /* Shouldn't ever get here unless we have something weird */
    }


/*
 * Perform a quick check to make sure that everything worked as it
 * should have.
 */

    switch(interface->bNumEndpoints) {
    case 2:
        if (!have_bulk_in || !have_bulk_out) {
            info("probe_scanner: Two bulk endpoints required.");
            return NULL;
        }
        break;
    case 3:
        if (!have_bulk_in || !have_bulk_out || !have_intr) {
            info("probe_scanner: Two bulk endpoints and one interrupt endpoint required.");
            return NULL;
        }
        break;
    default:
```

```
        info("probe_scanner: Endpoint determination failed.  Notify the maintainer.");
        return NULL;
    }


```

除控制端点外，扫描器的接口上还应该有输出和输入两个成块传输端点。此外可能还有一个中断传输端点。这里用到的几个宏操作均定义于 drivers/usb/scanner.h 中：

==================== drivers/usb/scanner.h 59 62 ====================

```
#define IS_EP_BULK(ep)  ((ep).bmAttributes == USB_ENDPOINT_XFER_BULK ? 1 : 0)
#define IS_EP_BULK_IN(ep) (IS_EP_BULK(ep) && ((ep).bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
#define IS_EP_BULK_OUT(ep) (IS_EP_BULK(ep) && ((ep).bEndpointAddress & USB_ENDPOINT_DIR_MASK) == USB_DIR_OUT)
#define IS_EP_INTR(ep) ((ep).bmAttributes == USB_ENDPOINT_XFER_INT ? 1 : 0)
```

每个端点都有个单字节的地址，其低 4 位为端点号，最高位则指明其方向，所以检查端点地址的最高位就可以知道块状端点的方向为输出或输入。当然，这个方向位对于双向的端点没有意义。此外，端点还有个属性字节，其最低两位表明端点的类型。有关的常数均定义于 include/linux/usb.h：

==================== include/linux/usb.h 72 79 ====================

```
#define USB_ENDPOINT_NUMBER_MASK 0x0f /* in bEndpointAddress */
#define USB_ENDPOINT_DIR_MASK         0x80

#define USB_ENDPOINT_XFERTYPE_MASK    0x03 /* in bmAttributes */
#define USB_ENDPOINT_XFER_CONTROL     0
#define USB_ENDPOINT_XFER_ISOC        1
#define USB_ENDPOINT_XFER_BULK        2
#define USB_ENDPOINT_XFER_INT         3
```

确认了扫描器的接口具有适当数量的端点以后，要为这个扫描器分配一个次设备号和一个 scn_usb_data 数据结构。这种数据结构定义于 drivers/usb/scanner.h：

==================== drivers/usb/scanner.h 87 99 ====================

```
struct scn_usb_data {
    struct usb_device *scn_dev;
    struct urb scn_irq;
    unsigned int ifnum;     /* Interface number of the USB device */
    kdev_t scn_minor;  /* Scanner minor - used in disconnect() */
    unsigned char button;  /* Front panel buffer */
      char isopen;          /* Not zero if the device is open */
    char present;      /* Not zero if device is present */
    char *obuf, *ibuf; /* transfer buffers */
    char bulk_in_ep, bulk_out_ep, intr_ep; /* Endpoint assignments */
    wait_queue_head_t rd_wait_q; /* read timeouts */
    struct semaphore gen_lock; /* lock to prevent concurrent reads or writes */
};
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构中的指针 obuf 和 ibuf 分别指向用于该扫描器的输出和输入缓冲区，所以要为之分配空间。

==================== drivers/usb/scanner.c 770 796 ====================
[usb_scanner_init()>usb_registcr()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>probe_scanner()]

```
/*
 * Determine a minor number and initialize the structure associated
 * with it.  The problem with this is that we are counting on the fact
 * that the user will sequentially add device nodes for the scanner
 * devices.  */

     for (scn_minor = 0; scn_minor < SCN_MAX_MNR; scn_minor++) {
          if (!p_scn_table[scn_minor])
               break;
     }

/* Check to make sure that the last slot isn't already taken */
     if (p_scn_table[scn_minor]) {
          err("probe_scanner: No more minor devices remaining.");
          return NULL;
     }

     dbg("probe_scanner: Allocated minor:%d", scn_minor);

     if (!(scn = kmalloc (sizeof (struct scn_usb_data), GFP_KERNEL))) {
          err("probe_scanner: Out of memory.");
          return NULL;
     }
     memset (scn, 0, sizeof(struct scn_usb_data));
     dbg ("probe_scanner(%d): Address of scn:%p", scn_minor, scn);

```

其它字段也需要根据已有的信息加以设置。特别地，如果设备具有中断端点，则要为这个端点调度一个周期性的中断传输，建立起对扫描器的定期查询（drivers/usb/scanner.c）。

==================== drivers/usb/scanner.c 797 841 ====================
[usb_scanner_init()>usb_registcr()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>probe_scanner()]

```
/* Ok, if we detected an interrupt EP, setup a handler for it */
     if (have_intr) {
          dbg("probe_scanner(%d): Configuring IRQ handler for intr EP:%d", scn_minor, have_intr);
          FILL_INT_URB(&scn->scn_irq, dev,
                    usb_rcvintpipe(dev, have_intr),
                    &scn->button, 1, irq_scanner, scn,
                    // endpoint[(int)have_intr].bInterval);
                    250);

            if (usb_submit_urb(&scn->scn_irq)) {
```

```
                err("probe_scanner(%d): Unable to allocate INT URB.", scn_minor);
                kfree(scn);
                return NULL;
            }
        }


/* Ok, now initialize all the relevant values */
        if (!(scn->obuf = (char *)kmalloc(OBUF_SIZE, GFP_KERNEL))) {
            err("probe_scanner(%d): Not enough memory for the output buffer.", scn_minor);
            kfree(scn);
            return NULL;
        }
        dbg("probe_scanner(%d): obuf address:%p", scn_minor, scn->obuf);

        if (!(scn->ibuf = (char *)kmalloc(IBUF_SIZE, GFP_KERNEL))) {
            err("probe_scanner(%d): Not enough memory for the input buffer.", scn_minor);
            kfree(scn->obuf);
            kfree(scn);
            return NULL;
        }
        dbg("probe_scanner(%d): ibuf address:%p", scn_minor, scn->ibuf);

        scn->bulk_in_ep = have_bulk_in;
        scn->bulk_out_ep = have_bulk_out;
        scn->intr_ep = have_intr;
        scn->present = 1;
        scn->scn_dev = dev;
        scn->scn_minor = scn_minor;
        scn->isopen = 0;

        init_MUTEX(&(scn->gen_lock));

        return p_scn_table[scn_minor] = scn;
}
```

我们把为扫描器调度中断传输的过程搁一下。暂且假定 probe_scanner()的操作已经完成，建立了扫描器的 usb_interface_descriptor 数据结构，并为之建立了 scn_usb_data 数据结构。回到 usb_find_interface_driver()的代码中，下一步是通过 usb_driver_claim_interface()正式“认领”这个设备（接口）（drivers/usb/usb.c）。

==================== drivers/usb/usb.c 461 477 ====================

[usb_scanner_init()>usb_registcr()>usb_scan_devices()>usb_check_support()>usb_find_interface_driver()>usb_driver_claim_interface()]

```
/*
 * This is intended to be used by usb device drivers that need to
 * claim more than one interface on a device at once when probing
 * (audio and acm are good examples).  No device driver should have
```

```
 * to mess with the internal usb_interface or usb_device structure
 * members.
 */
void usb_driver_claim_interface(struct usb_driver *driver, struct usb_interface *iface, void* priv)
{
	if (!iface || !driver)
		return;

	dbg("%s driver claimed interface %p", driver->name, iface);

	iface->driver = driver;
	iface->private_data = priv;
} /* usb_driver_claim_interface() */
```

总之，所谓“认领”就是使一个 usb_interface_descriptor 数据结构与相应设备的 usb_driver 结构挂上钩。这样，从具体设备的数据结构出发，就可以找到其设备驱动程序了。就这样，当 usb_scan_devices()完成了对所有 USB 设备的扫描时，对扫描器设备的登记和初始化就完成了。最后，以次设备号中的低 4 位为下标的指针数组 p_scn_table[]记录着指向每个具体 scn_usb_data 结构的地址。

8.9.4 USB 设备的驱动

要对一台扫描器操作时，首先也要通过系统调用 open()打开设备文件。所有的 USB 设备都有相同的主设备号 USB_MAJOR，而根据次设备号划分具体的设备及其类型。所以，根据设备文件节点提供的主设备号，CPU 首先找到 USB 总线的 file_operations 数据结构 usb_fops，从中得到用于 open 操作的函数指针，这个指针指向 usb_open()。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 2177 2201 ====================

```
static int usb_open(struct inode * inode, struct file * file)
{
	int minor = MINOR(inode->i_rdev);
	struct usb_driver *c = usb_minors[minor/16];
	int err = -ENODEV;
	struct file_operations *old_fops, *new_fops = NULL;

	/*
	 * No load-on-demand? Randy, could you ACK that it's really not
	 * supposed to be done?					-- AV
	 */
	if (!c || !(new_fops = fops_get(c->fops)))
		return err;
	old_fops = file->f_op;
	file->f_op = new_fops;
	/* Curiouser and curiouser... NULL ->open() as "no device" ? */
	if (file->f_op->open)
		err = file->f_op->open(inode,file);
```

```
2195     if (err) {
2196          fops_put(file->f_op);
2197          file->f_op = fops_get(old_fops);
2198     }
2199     fops_put(old_fops);
2200     return err;
2201 }
```

然后，进一步根据次设备号从指针数组 usb_minors[]中找到扫描器的 usb_driver 结构。我们知道，file 结构中的指针 f_op 应该指出向目标设备的 file_operations 数据结构，2188 行引用的 fops_get()是定义于 include/linux/fs.h 的宏操作：

==================== include/linux/fs.h 860 863 ====================

```
#define fops_get(fops) \
     (((fops) && (fops)->owner)  \
          ? ( try_inc_mod_count((fops)->owner) ? (fops) : NULL ) \
          : (fops))
```

扫描器的 file_operations 结构是 usb_scanner_fops，其函数指针 open 指向 open_scanner()，这个函数的代码在 drivers/usb/scanner.c 中：

==================== drivers/usb/scanner.c 345 402 ====================

[usb_open()>open_scanner()]

```
static int
open_scanner(struct inode * inode, struct file * file)
{
     struct scn_usb_data *scn;
     struct usb_device *dev;

     kdev_t scn_minor;

     int err=0;

     lock_kernel();

     scn_minor = USB_SCN_MINOR(inode);

     dbg("open_scanner: scn_minor:%d", scn_minor);

     if (!p_scn_table[scn_minor]) {
          err("open_scanner(%d): Unable to access minor data", scn_minor);
          err = -ENODEV;
          goto out_error;
     }

     scn = p_scn_table[scn_minor];

     dev = scn->scn_dev;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (!dev) {
        err("open_scanner(%d): Scanner device not present", scn_minor);
        err = -ENODEV;
        goto out_error;
    }

    if (!scn->present) {
        err("open_scanner(%d): Scanner is not present", scn_minor);
        err = -ENODEV;
        goto out_error;
    }

    if (scn->isopen) {
        err("open_scanner(%d): Scanner device is already open", scn_minor);
        err = -EBUSY;
        goto out_error;
    }

    init_waitqueue_head(&scn->rd_wait_q);

    scn->isopen = 1;

    file->private_data = scn; /* Used by the read and write metheds */

    MOD_INC_USE_COUNT;

out_error:

    unlock_kernel();

    return err;
}
```

我们把这个函数留给读者。有关 p_scn_table[]的作用可参阅前一小节中 probe_scanner()的代码。

打开了扫描器设备，就可以对其进行读 / 写，从此以后的操作有着两个层次上的内容。其一是取决于具体设备，即扫描器本身的操作，例如“开启光源”、“设置对比度”、“设置扫描精度”、“开始扫描”等等；其二是 USB 总线的操作，即如何实现主机与目标设备之间的信息传输，使对具体设备的操作得以实现。如果拿内核与应用软件之间的关系作一个比拟，则前者可以比作应用软件，而后者可以比作内核。前者是目的，后者是手段和基础设施。就一般的设备而言，这二者都是在内核中实现的，对 USB 设备的驱动当然也可以把这二者都放在内核中实现。可是，对一些速度要求不高、操作不很频繁、与内核中其它成分关系又不大的设备，把前者放在用户空间也有好处。拿扫描器来说，不同厂商、不同型号的扫描器在具体的操作上可能有相当的区别，放在用户空间中实现可以提供更大的灵活性。所以，对扫描器本身的操作是在进程这一层次上实现的，而内核只提供对扫描器操作的手段。我们在这里的目的并不在于扫描器本身，而在于驱动 USB 设备的手段及基础设施，就好像本书的题材不在于各种应用软件，而在于使这些应用软件得以运行的手段及基础设施一样。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

一般来说，从系统的角度来看，对设备的操作无非是两方面的信息交换，一方面是控制／状态信息的传输，另一方面是数据的传输，常常分别称为“控制通道”（control path）和“数据通道”（data path）。至于以何种方式实现这两种通道，则存在着一些灵活性。例如，对于扫描器的数据通道一般都是通过“成块”（bulk）方式传输的。这是因为对扫描器的输入显然没有实时要求，不需要通过“等时”（isochronous）方式传输。对于控制通道则更无采用等时传输的理由。但是，在采用控制传输还是成块传输方式来传递控制／状态信息这一点上，如果具体的设备允许并提供了相应的手段，却还是可以推敲的。扫描器就是这样，现代的扫描器大多支持“扫描器语言”，使得通过数据通道也可以实现控制目的（类似于终端设备的 Escape 序列）。如前所述，控制传输的优先级别比成块传输高，而且在 USB 总线上保留了一定的带宽用于控制传输，所以比较可靠。但是，若采用成块方式传输则可以简化程序设计，因为那样就可以不加区分地一律采用成块方式传输。从表面上看，这样会使控制／状态信息的传输降低优先级别并得不到保障，可是对于扫描器这样的设备其实并无多大区别。试想，如果 USB 总线的负荷已经大到不能保证传输数据的地步，那么提高控制／状态信息的优先级别，以此来保证能启动扫描还有多大意义？所以，在扫描器的设备驱动代码中有个条件编译控制 SCN_IOCTL。如果选择了这个选项，则应用进程须通过系统调用 ioctl()传递控制／状态信息，而通过 read()／write()读／写数据；否则便一律通过 read()／write()对扫描器操作。我们在这里假定选择通过 ioctl()传递控制／状态信息，目的是让读者由此可以看到控制传输的实例。扫描器的 ioctl()函数为 ioctl_scanner()。其代码在 drivers/usb/scanner.c 中：

==================== drivers/usb/scanner.c 863 940 ====================

```
#ifdef SCN_IOCTL
static int
ioctl_scanner(struct inode *inode, struct file *file,
         unsigned int cmd, unsigned long arg)
{
    struct usb_device *dev;

    int result;

    kdev_t scn_minor;

    scn_minor = USB_SCN_MINOR(inode);

    if (!p_scn_table[scn_minor]) {
        err("ioctl_scanner(%d): invalid scn_minor", scn_minor);
        return -ENODEV;
    }

    dev = p_scn_table[scn_minor]->scn_dev;

    switch (cmd)
    {
    case PV8630_IOCTL_INREQUEST :
    {
        struct {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                __u8  data;
                __u8  request;
                __u16 value;
                __u16 index;
            } args;

            if (copy_from_user(&args, (void *)arg, sizeof(args)))
                return -EFAULT;

            result = usb_control_msg(dev, usb_rcvctrlpipe(dev, 0),
                         args.request, USB_TYPE_VENDOR|
                         USB_RECIP_DEVICE|USB_DIR_IN,
                         args.value, args.index, &args.data,
                         1, HZ*5);

            dbg("ioctl_scanner(%d): inreq: args.data:%x args.value:%x args.index:%x args.request:%x\n", scn_minor, args.data, args.value, args.index, args.request);

            if (copy_to_user((void *)arg, &args, sizeof(args)))
                return -EFAULT;

            dbg("ioctl_scanner(%d): inreq: result:%d\n", scn_minor, result);

            return result;
        }
        case PV8630_IOCTL_OUTREQUEST :
        {
            struct {
                __u8  request;
                __u16 value;
                __u16 index;
            } args;

            if (copy_from_user(&args, (void *)arg, sizeof(args)))
                return -EFAULT;

            dbg("ioctl_scanner(%d): outreq: args.value:%x args.index:%x args.request:%x\n", scn_minor, args.value, args.index, args.request);

            result = usb_control_msg(dev, usb_sndctrlpipe(dev, 0),
                         args.request, USB_TYPE_VENDOR|
                         USB_RECIP_DEVICE|USB_DIR_OUT,
                         args.value, args.index, NULL,
                         0, HZ*5);

            dbg("ioctl_scanner(%d): outreq: result:%d\n", scn_minor, result);

            return result;
        }
```

```
935      default:
936          return -ENOIOCTLCMD;
937      }
938      return 0;
939  }
940  #endif /* SCN_IOCTL */
```

驱动程序的设计者为扫描器的 ioctl() 操作定义了 PV8630_IOCTL_INREQUEST 和 PV8630_IOCTL_OUTREQUEST 两种命令，前者用于从扫描器读入状态信息，后者用于向扫描器发出控制命令。二者都通过 usb_control_msg()完成信息的传递。比较一下 897 和 925 行，可以看出所不同的有两点。首先是第二个参数，一为 usb_rcvctrlpipe(dev, 0)，一为 usb_sndctrlpipe(dev, 0)；还有就是第 5 个参数中的标志位 USB_DIR_IN 和 USB_DIR_OUT。先看第二个参数，这里引用的两个宏操作均定义于 include/linux/usb.h：

==================== include/linux/usb.h 700 700 ====================

```
700  #define PIPE_CONTROL            2
```

==================== include/linux/usb.h 745 746 ====================

```
745  #define usb_sndctrlpipe(dev,endpoint)((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint))
746  #define usb_rcvctrlpipe(dev,endpoint)((PIPE_CONTROL << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
```

这里的__create_pipe()是个 inline 函数，也是在 include/linux/usb.h 中定义的：

==================== include/linux/usb.h 734 737 ====================

```
734  static inline unsigned int __create_pipe(struct usb_device *dev, unsigned int endpoint)
735  {
736      return (dev->devnum << 8) | (endpoint << 15) | (dev->slow << 26);
737  }
```

这些操作为本次传输构筑起一个 32 位长字，以后将用作传令（token）信包的主体。这个长字的最高两位表示传输的类型，中部是 7 位的设备地址加上 4 位的端点号。其 bit26 表示对方是否低速设备，bit7 为 1 则表示方向为输入。

常数 USB_DIR_OUT 和 USB DIR_IN 也定义于 include/linux/usb.h：

==================== include/linux/usb.h 38 42 ====================

```
38   /*
39    * USB directions
40    */
41   #define USB_DIR_OUT             0
42   #define USB_DIR_IN          0x80
```

此外，同一文件中为等时、成块以及中断管道也定义了相应的宏操作以及常数。

==================== include/linux/usb.h 747 752 ====================

```
747  #define usb_sndisocpipe(dev,endpoint)((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint))
748  #define usb_rcvisocpipe(dev,endpoint)((PIPE_ISOCHRONOUS << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
749  #define usb_sndbulkpipe(dev,endpoint)((PIPE_BULK << 30) | __create_pipe(dev,endpoint))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#define usb_rcvbulkpipe(dev,endpoint)((PIPE_BULK << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
#define usb_sndintpipe(dev,endpoint) ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint))
#define usb_rcvintpipe(dev,endpoint) ((PIPE_INTERRUPT << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
```

==================== include/linux/usb.h 698 701 ====================

```
#define PIPE_ISOCHRONOUS        0
#define PIPE_INTERRUPT          1
#define PIPE_CONTROL            2
#define PIPE_BULK          3
```

再看第 4 个参数，其中的 USB_RECIP_DEVICE 等常数也定义于 include/linux/usb.h 中：

==================== include/linux/usb.h 21 36 ====================

```
/*
 * USB types
 */
#define USB_TYPE_STANDARD       (0x00 << 5)
#define USB_TYPE_CLASS          (0x01 << 5)
#define USB_TYPE_VENDOR         (0x02 << 5)
#define USB_TYPE_RESERVED       (0x03 << 5)

/*
 * USB recipients
 */
#define USB_RECIP_MASK          0x1f
#define USB_RECIP_DEVICE        0x00
#define USB_RECIP_INTERFACE     0x01
#define USB_RECIP_ENDPOINT      0x02
#define USB_RECIP_OTHER         0x03
```

常数 USB_RECIP_DEVICE 表示传输的终极对象是设备，而不是接口，也不是端点本身。USB_TYPE_VENDOR 则表示在目标设备中要访问的寄存器是由设备制造商定义的，而不属于标准的 USB 寄存器。作为对比，读者可以回到前一小节看一下枚举阶段对 usb_control_msg()的调用，在那里第 4 个参数常常是 0 或 USB_DIR_IN，这就是因为 USB_TYPE_STANDARD 和 USB_RECIP_DEVICE 的定义都是 0，USB_DIR_OUT 也是 0。

虽然我们在这里是结合具体的设备扫描器读 usb_control_msg()的代码，这却是个通用的、属于基础设施一类的函数。在前一小节中，我们就看到在 USB 设备枚举的过程中频繁地调用这个函数从设备读取各种信息。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1065 1108 ====================

[ioctl_scanner()>usb_control_msg()]

```
/**
 *  usb_control_msg - Builds a control urb, sends it off and waits for completion
 *  @dev: pointer to the usb device to send the message to
 *  @pipe: endpoint "pipe" to send the message to
 *  @request: USB message request value
 *  @requesttype: USB message request type value
 *  @value: USB message value
```

```
 *   @index: USB message index value
 *   @data: pointer to the data to send
 *   @size: length in bytes of the data to send
 *   @timeout: time to wait for the message to complete before timing out (if 0 the wait is forever)
 *
 *   This function sends a simple control message to a specified endpoint
 *   and waits for the message to complete, or timeout.
 *
 *   If successful, it returns 0, othwise a negative error number.
 *
 *   Don't use this function from within an interrupt context, like a
 *   bottom half handler.  If you need a asyncronous message, or need to send
 *   a message from within interrupt context, use usb_submit_urb()
 */
int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8 requesttype,
               __u16 value, __u16 index, void *data, __u16 size, int timeout)
{
    devrequest *dr = kmalloc(sizeof(devrequest), GFP_KERNEL);
    int ret;

    if (!dr)
         return -ENOMEM;

    dr->requesttype = requesttype;
    dr->request = request;
    dr->value = cpu_to_le16p(&value);
    dr->index = cpu_to_le16p(&index);
    dr->length = cpu_to_le16p(&size);

    //dbg("usb_control_msg");

    ret = usb_internal_control_msg(dev, pipe, dr, data, size, timeout);

    kfree(dr);

    return ret;
}
```

参数 dev 指向目标设备的 usb_device 数据结构。pipe 是个 32 位无符号整数，那就是上面构筑的 32 位长字。其最高两位表示传输的类型（等时 / 中断 / 控制 / 成块），其余各位包括对方的端点号以及设备号，以及设备是否为全速（或低速）。此外，requesttype 是个 8 位字节，其最高位表示传输的方向，最低 5 位则表明传输终极对象的类别（设备 / 接口 / 端点 / 其它），通常这意味着一组寄存器或者一小块存储区间；而 index 则进而指明了具体的单元，这就是终极的操作对象。针对这个操作对象，request 说明了需要进行的操作，而 value 则是参数。如果有更多的数据需要传递（读 / 写），则通过缓冲区 data 进行，其大小为 size。这些都是从用户空间传下的参数，而传输的目的正是要把这些信息发送给目标设备。最后，参数 timeout 表示愿意睡眠等待传输完成的时间。显然，usb_control_msg()只是个外包装，实际的操作由 usb_internal_control_msg()完成。在传输的过程中，端点起着类似于“传达室”的作用。

根据 USB 总线规格书的规定，需要发送给目标设备的这些信息要组装在一个操作请求块中。这里的 devrequest 就是根据这个规定而定义的数据结构，其定义在 include/linux/usb.h 中：

==================== include/linux/usb.h 151 157 ====================

```
typedef struct {
        __u8 requesttype;
        __u8 request;
        __u16 value;
        __u16 index;
        __u16 length;
} devrequest __attribute__ ((packed));
```

实际上，这个数据结构就是 SETUP 信包的内容，而缓冲区的内容，则就是随后的数据信包的内容。至于状态阶段的信包，则是来自（或去向）终极操作对象（而不是交互对象）的确认。

函数 usb_internal_control_msg()的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1042 1063 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()]

```
// returns status (negative) or length (positive)
int usb_internal_control_msg(struct usb_device *usb_dev, unsigned int pipe,
                      devrequest *cmd,  void *data, int len, int timeout)
{
     urb_t *urb;
     int retv;
     int length;

     urb = usb_alloc_urb(0);
     if (!urb)
          return -ENOMEM;

     FILL_CONTROL_URB(urb, usb_dev, pipe, (unsigned char*)cmd, data, len,    /* build urb */
            (usb_complete_t)usb_api_blocking_completion,0);

     retv = usb_start_wait_urb(urb, timeout, &length);
     if (retv < 0)
          return retv;
     else
          return length;

}
```

对于 USB 总线上的每个传输，需要为之创建一个“ USB 传输请求块”，即 usb 数据结构。简而言之，发送控制报文的过程就是：根据参数建立一个 usb 数据结构，把这个 usb 结构交给低层，让低层据以调度相应的控制传输，然后睡眠等待传输的完成。

这种数据结构的定义在 include/linux/usb.h 中：

==================== include/linux/usb.h 440 466 ====================

```
typedef struct urb
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
    spinlock_t lock;        // lock for the URB
    void *hcpriv;           // private data for host controller
    struct list_head urb_list;  // list pointer to all active urbs
    struct urb *next;       // pointer to next URB
    struct usb_device *dev;     // pointer to associated USB device
    unsigned int pipe;      // pipe information
    int status;             // returned status
    unsigned int transfer_flags;     // USB_DISABLE_SPD | USB_ISO_ASAP | etc.
    void *transfer_buffer;      // associated data buffer
    int transfer_buffer_length; // data buffer length
    int actual_length;            // actual data buffer length
    int bandwidth;              // bandwidth for this transfer request (INT or ISO)
    unsigned char *setup_packet;     // setup packet (control only)
    //
    int start_frame;        // start frame (iso/irq only)
    int number_of_packets;      // number of packets in this request (iso)
    int interval;                // polling interval (irq only)
    int error_count;        // number of errors in this transfer (iso only)
    int timeout;            // timeout (in jiffies)
    //
    void *context;              // context for completion routine
    usb_complete_t complete;    // pointer to completion routine
    //
    iso_packet_descriptor_t iso_frame_desc[0];
} urb_t, *purb_t;
```

同一文件中还定义了一个宏操作 FILL_CONTROL_URB():

==================== include/linux/usb.h 468 478 ====================

```
#define FILL_CONTROL_URB(a,aa,b,c,d,e,f,g) \
   do {\
    spin_lock_init(&(a)->lock);\
    (a)->dev=aa;\
    (a)->pipe=b;\
    (a)->setup_packet=c;\
    (a)->transfer_buffer=d;\
    (a)->transfer_buffer_length=e;\
    (a)->complete=f;\
    (a)->context=g;\
   } while (0)
```

对照上面对 FILL_CONTROL_URB()的引用，可知 urb->dev 设置成指向 usb_dev，urb->pipe 为目标设备的控制管道，即端点 0，urb->setup_packet 指向前面创建的 devrequest 数据结构，urb->transfer_buffer 设置成前面的参数 data，urb->complete 指向 usb_api_blocking_completion()，urb->context 则暂时为 0。

设置好 urb 数据结构以后，就通过 usb_start_wait_urb()“提交”这个 urb 数据结构，并等待交互的完成。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 995 1039 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()]

```
// Starts urb and waits for completion or timeout
static int usb_start_wait_urb(urb_t *urb, int timeout, int* actual_length)
{
     DECLARE_WAITQUEUE(wait, current);
     DECLARE_WAIT_QUEUE_HEAD(wqh);
     api_wrapper_data awd;
     int status;

     awd.wakeup = &wqh;
     init_waitqueue_head(&wqh);
     current->state = TASK_INTERRUPTIBLE;
     add_wait_queue(&wqh, &wait);
     urb->context = &awd;
     status = usb_submit_urb(urb);
     if (status) {
          // something went wrong
          usb_free_urb(urb);
          current->state = TASK_RUNNING;
          remove_wait_queue(&wqh, &wait);
          return status;
     }

     if (urb->status == -EINPROGRESS) {
          while (timeout && urb->status == -EINPROGRESS)
               status = timeout = schedule_timeout(timeout);
     } else
          status = 1;

     current->state = TASK_RUNNING;
     remove_wait_queue(&wqh, &wait);

     if (!status) {
          // timeout
          printk("usb_control/bulk_msg: timeout\n");
          usb_unlink_urb(urb);  // remove urb safely
          status = -ETIMEDOUT;
     } else
          status = urb->status;

     if (actual_length)
          *actual_length = urb->actual_length;

     usb_free_urb(urb);
     return status;
}
```

像以前多次见到的那样（详见第 4 章），在当前进程的系统空间堆栈上建立起一个等待队列，并将当前进程的 task 结构通过一个 wait_queue_t 数据结构挂在队列中。同时，又定义了一个 api_wrapper_data

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

数据结构 awd。这种数据结构定义于 include/linux/usb.h：

==================== include/linux/usb.h 540 546 ====================

```
typedef struct
{
  wait_queue_head_t *wakeup;

  void* stuff;
  /* more to follow */
} api_wrapper_data;
```

使 awd.wakeup 指向上述的队列头，再使 urb 结构中的指针 context 指向 awd，就在 urb 结构与等待队列之间建立起了联系。这样，从具体的 urb 结构出发，就可以找到正在等待其（所代表的传输）完成的进程。然后就通过 usb_submit_urb()提交这个传输请求。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 955 961 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()]

```
int usb_submit_urb(urb_t *urb)
{
     if (urb && urb->dev)
          return urb->dev->bus->op->submit_urb(urb);
     else
          return -ENODEV;
}
```

这里使用了“提交”（submit）这个词，但并不是向上层提交，相反倒是交付给下层的意思。那么下层是什么呢？这就是 UHCI 或 OHCI。对于采用 UHCI 界面的 USB 总线控制器，其 usb_bus 结构中的 usb_operations 结构指针 op 指向 drivers/usb/uhci.c 中定义的 uhci_device_operations。

==================== drivers/usb/uhci.c 1615 1621 ====================

```
struct usb_operations uhci_device_operations = {
     uhci_alloc_dev,
     uhci_free_dev,
     uhci_get_current_frame_number,
     uhci_submit_urb,
     uhci_unlink_urb
};
```

显然，UHCI 界面的函数指针 submit_urb 指向 uhci_submit_urb()。这是个通用的函数，不但用来交付控制传输，也用来交付等时、中断以及成块传输。其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1291 1377 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()
>uhci_submit_urb()]

```
static int uhci_submit_urb(struct urb *urb)
{
     int ret = -EINVAL;
     struct uhci *uhci;
     unsigned long flags;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	struct urb *u;
	int bustime;

	if (!urb)
		return -EINVAL;

	if (!urb->dev || !urb->dev->bus || !urb->dev->bus->hcpriv)
		return -ENODEV;

	uhci = (struct uhci *)urb->dev->bus->hcpriv;

	/* Short circuit the virtual root hub */
	if (usb_pipedevice(urb->pipe) == uhci->rh.devnum)
		return rh_submit_urb(urb);

	u = uhci_find_urb_ep(uhci, urb);
	if (u && !(urb->transfer_flags & USB_QUEUE_BULK))
		return -ENXIO;

	usb_inc_dev_use(urb->dev);
	spin_lock_irqsave(&urb->lock, flags);

	if (!uhci_alloc_urb_priv(urb)) {
		spin_unlock_irqrestore(&urb->lock, flags);
		usb_dec_dev_use(urb->dev);

		return -ENOMEM;
	}

	switch (usb_pipetype(urb->pipe)) {
	case PIPE_CONTROL:
		ret = uhci_submit_control(urb);
		break;
	case PIPE_INTERRUPT:
		if (urb->bandwidth == 0) {	/* not yet checked/allocated */
			bustime = usb_check_bandwidth(urb->dev, urb);
			if (bustime < 0)
				ret = bustime;
			else {
				ret = uhci_submit_interrupt(urb);
				if (ret == -EINPROGRESS)
					usb_claim_bandwidth(urb->dev, urb, bustime, 0);
			}
		} else		/* bandwidth is already set */
			ret = uhci_submit_interrupt(urb);
		break;
	case PIPE_BULK:
		ret = uhci_submit_bulk(urb, u);
		break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	case PIPE_ISOCHRONOUS:
		if (urb->bandwidth == 0) {	/* not yet checked/allocated */
			if (urb->number_of_packets <= 0) {
				ret = -EINVAL;
				break;
			}
			bustime = usb_check_bandwidth(urb->dev, urb);
			if (bustime < 0) {
				ret = bustime;
				break;
			}

			ret = uhci_submit_isochronous(urb);
			if (ret == -EINPROGRESS)
				usb_claim_bandwidth(urb->dev, urb, bustime, 1);
		} else		/* bandwidth is already set */
			ret = uhci_submit_isochronous(urb);
		break;
	}

	urb->status = ret;

	spin_unlock_irqrestore(&urb->lock, flags);

	if (ret == -EINPROGRESS)
		ret = 0;
	else {
		uhci_unlink_generic(urb);
		usb_dec_dev_use(urb->dev);
	}

	return ret;
}
```

这是个需要细加阅读的函数。

对于采用 UHCI 界面的 USB 总线控制器，其 usb_bus 结构中的指针 hcpriv 指向一个 uhci 数据结构，而 uhci 结构中有个次层结构 rh，里面是有关根集中器的信息，这是在根集中器初始化时设置好的。如果目标设备恰好就是根集中器，那么通信的过程可以简化，因为 CPU 直接就可以访问其各个寄存器，不需要通过 USB 总线就能进行通信，所以此时由 rh_submit_urb()完成操作。这个函数的代码在 drivers/usb/uhci.c 中，但是我们在这里就从略了。

与其它所有 USB 设备的通信（传输）都要通过 USB 总线上的交互来完成，因而需要为具体的传输调度一个或几个交互。除等时传输之外，在调度中不允许同时存在对同一对象的两个同种传输。这是为什么呢？我们以前讲过，控制传输和成块传输是以传输（而不是交互）为单位挂入调度队列的，一个传输就是一个交互队列，而对这些队列的执行又可以是横向执行。如果对同一对象的两个传输都挂在调度队列中，就有可能使两个传输中的交互夹杂在一起；例如从第一个传输中执行了一个交互以后就在第二个传输中也执行一个交互，然后又回到第一个传输，这当然就乱了套。至于中断传输，则

PDF-XCHANGE Click to buy NOW! www.docu-track.com

只包含一个交互，并且是以交互为单位进行调度的，但是却并没有理由对同一端点调度两个中断传输。

所以，先通过 uhci_find_urb_ep()，寻找已经调度而尚未完成的对同一对象的同种传输，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1263 1289 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_find_urb_ep()]

```
1263 static struct urb *uhci_find_urb_ep(struct uhci *uhci, struct urb *urb)
1264 {
1265     struct list_head *tmp, *head = &uhci->urb_list;
1266     unsigned long flags;
1267     struct urb *u = NULL;
1268
1269     if (usb_pipeisoc(urb->pipe))
1270         return NULL;
1271
1272     nested_lock(&uhci->urblist_lock, flags);
1273     tmp = head->next;
1274     while (tmp != head) {
1275         u = list_entry(tmp, struct urb, urb_list);
1276
1277         tmp = tmp->next;
1278
1279         if (u->dev == urb->dev &&
1280             u->pipe == urb->pipe)
1281              goto found;
1282     }
1283     u = NULL;
1284
1285 found:
1286     nested_unlock(&uhci->urblist_lock, flags);
1287
1288     return u;
1289 }
```

等时交互的调度是个不同的问题，所以如果对方是个等时端点（1269 行）就立即返回 0。这里的 usb_pipeisoc()是个宏操作，定义于 include/linux/usb.h。与此类似的宏操作还有 usb_pipeint()、usb_pipecontrol()以及 usb_pipebulk()。

==================== include/linux/usb.h 716 719 ====================

```
#define usb_pipeisoc(pipe)  (usb_pipetype((pipe)) == PIPE_ISOCHRONOUS)
#define usb_pipeint(pipe)   (usb_pipetype((pipe)) == PIPE_INTERRUPT)
#define usb_pipecontrol(pipe)    (usb_pipetype((pipe)) == PIPE_CONTROL)
#define usb_pipebulk(pipe)  (usb_pipetype((pipe)) == PIPE_BULK)
```

代码中的 nested_lock()和 nested_unlock()也是 drivers/usb/uhci.h 中定义的宏操作：

==================== drivers/usb/uhci.h 21 35 ====================

```
#define nested_lock(snl, flags) \
```

```
    if ((snl)->uniq == current) { \
        (snl)->count++; \
        flags = 0; /* No warnings */ \
    } else { \
        spin_lock_irqsave(&(snl)->lock, flags); \
        (snl)->count++; \
        (snl)->uniq = current; \
    }

#define nested_unlock(snl, flags) \
    if (!--(snl)->count) { \
        (snl)->uniq = NULL; \
        spin_unlock_irqrestore(&(snl)->lock, flags); \
    }
```

如果在队列 urb_list 中找到了目标设备相同，而且端点也相同的 urb 结构，就返回指向该结构的指针，否则返回 NULL。一般而言，如果已经存在对同一目标的同种传输，就应该拒绝调度新的传输请求，让高层的软件决定是否稍后再来试试。其实，控制传输都是很短暂的，并且启动控制传输的进程要睡眠等待其完成，所以发生这种情况的可能性本来就很小。但是，对于成块传输则情况有所不同，所以允许将新的传输与已经存在的传输合并，但是必须将对同一对象的传输“串行化”，就是把它们的交互依次连接在同一个队列中。这样，不管是横向执行还是纵向执行都能保证正确的次序。

除 urb 数据结构以外，还需要有个 urb_priv 数据结构与其配合使用。这种数据结构定义于 drivers/usb/uhci.h：

==================== drivers/usb/uhci.h 337 354 ====================

```
struct urb_priv {
    struct urb *urb;

    struct uhci_qh *qh;         /* QH for this URB */

    int fsbr : 1;           /* URB turned on FSBR */
    int fsbr_timeout : 1;       /* URB timed out on FSBR */
    int queued : 1;             /* QH was queued (not linked in) */
    int short_control_packet : 1;    /* If we get a short packet during */
                        /*  a control transfer, retrigger */
                        /*  the status phase */

    unsigned long inserttime;   /* In jiffies */

    struct list_head list;

    struct list_head urb_queue_list; /* URB's linked together */
};
```

函数 uhci_alloc_urb_priv()的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 480 499 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()

>uhci_submit_urb()>uhci_alloc_urb_priv()]

```
struct urb_priv *uhci_alloc_urb_priv(struct urb *urb)
{
     struct urb_priv *urbp;

     urbp = kmem_cache_alloc(uhci_up_cachep, in_interrupt() ? SLAB_ATOMIC : SLAB_KERNEL);
     if (!urbp)
          return NULL;

     memset((void *)urbp, 0, sizeof(*urbp));

     urbp->inserttime = jiffies;
     urbp->urb = urb;

     INIT_LIST_HEAD(&urbp->list);
     INIT_LIST_HEAD(&urbp->urb_queue_list);

     urb->hcpriv = urbp;

     return urbp;
}
```

下面的操作，就取决于传输的类型，也就是目标端点的类型（1325 行）了。我们先结合 usb_control_msg()的情景看控制传输的调度，以后还要再回过来看其它几种传输。

控制传输是通过 uhci_submit_control()调度和实现的，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 633 741 ====================
[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()
>uhci_submit_urb()>uhci_submit_control()]

```
/*
 * Control transfers
 */
static int uhci_submit_control(struct urb *urb)
{
     struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
     struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
     struct uhci_td *td;
     struct uhci_qh *qh;
     unsigned long destination, status;
     int maxsze = usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe));
     int len = urb->transfer_buffer_length;
     unsigned char *data = urb->transfer_buffer;

     /* The "pipe" thing contains the destination in bits 8--18 */
     destination = (urb->pipe & PIPE_DEVEP_MASK) | USB_PID_SETUP;

     /* 3 errors */
     status = (urb->pipe & TD_CTRL_LS) | TD_CTRL_ACTIVE | (3 << 27);

```

```
	/*
	 * Build the TD for the control request
	 */
	td = uhci_alloc_td(urb->dev);
	if (!td)
		return -ENOMEM;

	uhci_add_td_to_urb(urb, td);
	uhci_fill_td(td, status, destination | (7 << 21),
		virt_to_bus(urb->setup_packet));

	/*
	 * If direction is "send", change the frame from SETUP (0x2D)
	 * to OUT (0xE1). Else change it from SETUP to IN (0x69).
	 */
	destination ^= (USB_PID_SETUP ^ usb_packetid(urb->pipe));

	if (!(urb->transfer_flags & USB_DISABLE_SPD))
		status |= TD_CTRL_SPD;

	/*
	 * Build the DATA TD's
	 */
	while (len > 0) {
		int pktsze = len;

		if (pktsze > maxsze)
			pktsze = maxsze;

		td = uhci_alloc_td(urb->dev);
		if (!td)
			return -ENOMEM;

		/* Alternate Data0/1 (start with Data1) */
		destination ^= 1 << TD_TOKEN_TOGGLE;

		uhci_add_td_to_urb(urb, td);
		uhci_fill_td(td, status, destination | ((pktsze - 1) << 21),
			virt_to_bus(data));

		data += pktsze;
		len -= pktsze;
	}

	/*
	 * Build the final TD for control status
	 */
	td = uhci_alloc_td(urb->dev);
	if (!td)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                return -ENOMEM;

        /*
         * It's IN if the pipe is an output pipe or we're not expecting
         * data back.
         */
        destination &= ~TD_PID;
        if (usb_pipeout(urb->pipe) || !urb->transfer_buffer_length)
                destination |= USB_PID_IN;
        else
                destination |= USB_PID_OUT;

        destination |= 1 << TD_TOKEN_TOGGLE;      /* End in Data1 */

        status &= ~TD_CTRL_SPD;

        uhci_add_td_to_urb(urb, td);
        uhci_fill_td(td, status | TD_CTRL_IOC,
                destination | (UHCI_NULL_DATA_SIZE << 21), 0);

        qh = uhci_alloc_qh(urb->dev);
        if (!qh)
                return -ENOMEM;

        /* Low speed or small transfers gets a different queue and treatment */
        if (urb->pipe & TD_CTRL_LS) {
                uhci_insert_tds_in_qh(qh, urb, 0);
                uhci_insert_qh(uhci, &uhci->skel_ls_control_qh, qh);
        } else {
                uhci_insert_tds_in_qh(qh, urb, 1);
                uhci_insert_qh(uhci, &uhci->skel_hs_control_qh, qh);
                uhci_inc_fsbr(uhci, urb);
        }

        urbp->qh = qh;

        uhci_add_urb_list(uhci, urb);

        return -EINPROGRESS;
}
```

一次控制传输至少要由两个交互完成。第一个交互是由主机通过 USB 总线控制器向目标设备的控制端点发送一个 SETUP 报文，这就是所谓的 SETUP 阶段。然后，根据需要传输的数据量大小而有一个或几个数据交互，每个交互传递一个 DATA 报文，传递的方向既可以是输出也可以是输入，这就是数据阶段。不过，如果没有数据就不需要传送 DATA 报义。最后，还要有一个状态交互，让 DATA 报文的接收者向对方发送一个状态报文，确认对 SETUP 报文或 DATA 报文的接收，这就是状态阶段。这样，最少是两个交互，通常则是三个交互。交互是 USB 控制器的执行单位。每个交互都由一个 uhci_td

PDF-XCHANGE Click to buy NOW! www.docu-track.com

数据结构代表，我们已经在前面看到过 uhci_td 数据结构的定义。如前所述，uhci_td 数据结构中的前 4 个 32 位长字是为 USB 控制器的硬件准备的，实际上相当于 4 个寄存器，作用分别如下：

(1) link。相当于“链接寄存器”。其高 28 位后面添上 4 位 0 成为一个指针，指向下一个 uhci_td 结构或 uhci_qh 结构的物理地址。其低 4 位则为标志位，其中 bit0 为 1 表示链接的终结，bit1 为 1 表示指针所指为队列头，即 uhci_qh 结构，否则为 uhci_td 结构。至于 bit2，则为 1 时表示纵向执行，为 0 时表示横向执行。

(2) status。相当于“控制 / 状态寄存器”。其中的 TD_CTRL_ACTIVE 状态位起着特殊的作用，为 1 时表示等待执行，为 0 时则表示已经执行。此外，寄存器中还有个重发次数位段，表示在出错的情况下允许重发几次。代码中（651 行）把重发次数设置成 3。USB 控制器在每一次出错重发时都把这个计数减 1，如果达到了 0 就认为彻底失败了。

(3) info。相当于“命令寄存器”，其内容包括传输的类型及目标，实际上就是 token 信包的主体。

(4) buffer。相当于“缓冲区地址寄存器”，指向发送 / 接收数据的缓冲区。

每个信包的头部都包含着一个 8 位的“信包标识码 ” PID，表明具体信包的类型。SETUP 信包的 PID 就是 USB_PID_SETUP，定义于 include/linux/usb.h：

==================== include/linux/usb.h 81 100 ====================

```
/*
 * USB Packet IDs (PIDs)
 */
#define USB_PID_UNDEF_0                         0xf0
#define USB_PID_OUT                            0xe1
#define USB_PID_ACK                            0xd2
#define USB_PID_DATA0                          0xc3
#define USB_PID_PING                           0xb4/* USB 2.0 */
#define USB_PID_SOF                            0xa5
#define USB_PID_NYET                           0x96/* USB 2.0 */
#define USB_PID_DATA2                          0x87     /* USB 2.0 */
#define USB_PID_SPLIT                          0x78     /* USB 2.0 */
#define USB_PID_IN                             0x69
#define USB_PID_NAK                            0x5a
#define USB_PID_DATA1                          0x4b
#define USB_PID_PREAMBLE                        0x3c     /* Token mode */
#define USB_PID_ERR                            0x3c/* USB 2.0: handshake mode */
#define USB_PID_SETUP                          0x2d
#define USB_PID_STALL                          0x1e
#define USB_PID_MDATA                          0x0f     /* USB 2.0 */
```

这些定义来自 USB 的规格书，其中有些是为 USB 2.0 定义的。

代码中先为 SETUP 交互分配一个 uhci_td 数据结构（656 行），同时还要为报文准备下 destination 和 status 两个 32 位字段。前者用于“命令寄存器”，其内容就是 token 信包的主体，包含着交互对象的地址和端点号，还有 SETUP 信包的 PID，即 USB_PID_SETUP。此外，信包的长度也组装在这个长字中，SETUP 信包的实际长度为 8 个字节（7+1）。

对于 SETUP 交互，要发送的报文是对目标设备的操作命令，这就是前面准备好的 devrequest 数据

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

结构，urb->setup_packet 就指向这个结构。

如上所述，“控制 / 状态寄存器”中的 TD_CTRL_ACTIVE 状态位起着特殊的作用，驱动软件在调度一个交互请求时将这一位设成 1，表示这是个待执行的“活跃”交互；而 USB 控制器，则在完成了该次交互以后（或成功，或彻底失败）就将这一位改成 0。这样，驱动软件只要扫描各个 uhci_td 数据结构，发现某个 uhci_td 数据结构的 TD_CTRL_ACTIVE 位变成了 0，就说明这个交互已经完成。

分配了一个 uhci_td 数据结构以后，就通过 uhci_add_td_to_urb()把它链入 urb_priv 结构中的交互描述块队列（660 行），并通过 inline 函数 uhci_fill_td()设置 uhci_td 结构中的几个重要字段（见前）。这样，就相当于设置好了 USB 控制器的 4 个寄存器。数据信包的内容来自缓冲区 urb->setup_packet，交互描述块中的字段 buffer 设置成缓冲区的物理地址；在发送数据信包时，USB 控制器就通过 DMA 从这个缓冲区中读取数据。准备好了一个交互请求，就通过 uhci_add_td_to_urb()，将其挂入传输请求即 urb 结构内的队列中（实际上是在 urb_priv 结构内）。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 501 508 ====================
[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_control()>uhci_add_td_to_urb()]

```
static void uhci_add_td_to_urb(struct urb *urb, struct uhci_td *td)
{
    struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;

    td->urb = urb;

    list_add_tail(&td->list, &urbp->list);
}
```

至此，已经为控制传输的第一阶段，即 SETUP 阶段准备好了需要传输的报文。USB 控制器在执行时将首先通过 token 信包在 USB 总线上发布一个通告，说明传输的类型（控制）、交互的对象（扫描器的控制端点）、信包的种类（SETUP）。然后，就会把前面准备好的 devrequest 数据结构作为 SETUP 信包，即第一阶段的数据发送出去。这个信包的内容告诉扫描器想要干什么，例如对扫描器内的哪一个寄存器进行何种操作。由于具体的寄存器是由扫描器的制造商定义的，这些内容对于 USB 总线是透明的。而扫描器，则在接收到这个 SETUP 信包以后要发回一个 ACK 信包加以确认。

如果所要求的操作需要有附加的数据传递，便有 DATA 报文要传送，但是其类型不再是 USB_PID_SETUP，而是 USB_PID_IN 或 USB_PID_OUT 了。所以一方面把 destination 中的 USB_PID_SETUP 去除，另一方面根据所用的管道确定是 USB_PID_IN 还是 USB_PID_OUT。这里的宏操作 usb_packetid()定义于 include/linux/usb.h：

include/linux/usb.h

```
#define usb_packetid(pipe)  (((pipe) & USB_DIR_IN) ? USB_PID_IN : USB_PID_OUT)
```

在 SETUP 报文以后有几个 DATA 信包取决于数据量和允许的信包大小。当有多个 DATA 信包需要传送时，需要给每个信包编上序号，使接收方能知道是否丢失了信包。为此目的，在 DATA 信包的头部采用了一个二进制位作为序号，其位置为 TD_TOKEN_TOGGLE，每（调度）发送一个信包就将这一位翻转一次（687 行），使序号为 0 与为 1 的 DATA 信包相间。因此，DATA 信包又可以分成 DATA0 和 DATA1 两种。如果接收方连续收到两个序号相同的信包，就知道一定是丢失了一个信包。当然，这

是建立在不会接连丢失两个信包这么个前提下的。

最后是一次“状态”（status）交互，由接收方发送一个确认报文。所以，如果整个传输（数据信包）是输出，那么确认信包就应该由设备向主机发送，所以对主机而言是 USB_PID_IN。或者，如果根本就没有数据信包，那也应该是 USB_PID_IN；否则就是 USB_PID_OUT。此外，确认信包的序号总是 1（714 行），没有数据部分（720 行），也不需要检测信包的长度。由于状态交互是控制传输中的最后一个交互，这里（719 行）还把标志位 TD_CTRL_IOC 设成 1，表示这个交互完成后要向 CPU 发出一个中断请求。注意不要将状态交互与握手信包相混淆，状态交互同样包括传令、数据和握手三个信包的传递。

这样，一个传输的所有交互描述块就都挂入了 urb 数据结构内的队列中。可是，这并不是最终的目的，最终的目的是要挂入 USB 总线的调度队列中。可是，这里有个问题。USB 总线的调度队列是供 USB 控制器硬件在“执行”时使用的，链接指针必须采用物理地址。可是，采用了物理地址，CPU 就不能顺着链接指针依次访问队列中的各个成分了。反过来，如果要考虑 CPU 的访问以及队列操作，就要采用虚拟地址；而若采用了虚拟地址，USB 控制器硬件就不能顺着链接依次执行了。对于具有“智能 DMA”功能的设备，这是个共同的问题。解决的办法是让采用虚拟地址和物理地址的链接手段在数据结构中同时并存。所以，在 uhci_td 数据结构的设计中就考虑到了这两方面的需要。里面既有常规的（虚拟地址）队列头 list，让 CPU 可以对 uhci_td 数据结构进行常规的队列操作；又有硬件所需的物理地址指针 link。前面的 uhci_add_td_to_urb()已经将所有的 uhci_td 数据结构通过它们的队列头 list 连成了常规的队列，下面就要再将这些 uhci_td 数据结构通过物理地址指针 link 链接起来。另一方面，控制传输是作为一个整体，即以 uhci_qh 数据结构为队列头的交互描述块队列挂入调度系统的，所以先要通过 uhci_alloc_qh()为之分配一个 uhci_qh 数据结构作为队列头部。其代码见 drivers/usb/uhci.c。

==================== drivers/usb/uhci.c 302 321 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_control()>uhci_alloc_qh()]

```
static struct uhci_qh *uhci_alloc_qh(struct usb_device *dev)
{
	struct uhci_qh *qh;

	qh = kmem_cache_alloc(uhci_qh_cachep, in_interrupt() ? SLAB_ATOMIC : SLAB_KERNEL);
	if (!qh)
		return NULL;

	qh->element = UHCI_PTR_TERM;
	qh->link = UHCI_PTR_TERM;

	qh->dev = dev;
	qh->prevqh = qh->nextqh = NULL;

	INIT_LIST_HEAD(&qh->remove_list);

	usb_inc_dev_use(dev);

	return qh;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

然后，就要看对方是低速设备还是全速设备了，因为二者在 uhci 数据结构中有不同的控制交互队列。不过，二者的操作基本上是相同的，我们在这里只看全速设备的情景。首先，通过 uhci_insert_tds_in_qh()将所有的 uhci_td 数据结构通过物理地址链接成一个队列，而上面分配的 uhci_qh 数据结构就是队列的头。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 302 321 ====================
[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_control()>uhci_insert_tds_in_qh()]

```
/*
 * Inserts a td into qh list at the top.
 */
static void uhci_insert_tds_in_qh(struct uhci_qh *qh, struct urb *urb, int breadth)
{
	struct list_head *tmp, *head;
	struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
	struct uhci_td *td, *prevtd;

	if (!urbp)
		return;

	head = &urbp->list;
	tmp = head->next;
	if (head == tmp)
		return;

	td = list_entry(tmp, struct uhci_td, list);

	/* Add the first TD to the QH element pointer */
	qh->element = virt_to_bus(td) | (breadth ? 0 : UHCI_PTR_DEPTH);

	prevtd = td;

	/* Then link the rest of the TD's */
	tmp = tmp->next;
	while (tmp != head) {
		td = list_entry(tmp, struct uhci_td, list);

		tmp = tmp->next;

		prevtd->link = virt_to_bus(td) | (breadth ? 0 : UHCI_PTR_DEPTH);

		prevtd = td;
	}

	prevtd->link = UHCI_PTR_TERM;
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

在 uhci_qh 结构中有两个采用物理地址的指针。一个是 link，用于队列之间的链接，称为横向链接；另一个则是 element，指向本队列中的第一个 uhci_td 结构，这是纵向链接。从代码中可见，首先将队列头的指针 element 设置成第一个 uhci_td 结构的物理地址，然后就依次使各个 uhci_td 结构中的指针 link 设置成下一个 uhci_td 结构的物理地址。不过，无论是 uhci_qh 结构还是 uhci_td 结构的地址都是与 16 字节边界对齐的，所以这些指针中的低 4 位都可以用作控制位。drivers/usb/usb-uhci.h 中定义了这些控制位：

==================== drivers/usb/usb-uhci.h 72 75 ====================

```
#define UHCI_PTR_BITS       0x000F
#define UHCI_PTR_TERM       0x0001
#define UHCI_PTR_QH         0x0002
#define UHCI_PTR_DEPTH      0x0004
```

其中之一就是 UHCI_PTR_DEPTH，定义为 4，即 bit2。当这个控制位为 1 时，表示 USB 控制器在执行时应该深度优先，即纵向执行；否则即为宽度优先，即横向执行。所以，对于低速设备因参数 breadth 为 0 而纵向执行；对全速设备则参数 breadth 为 1 而横向执行。此外，队列中最后一个 uhci_td 结构的指针 link 设置成 UHCI_PTR_TERM，就是地址为 0，而最低位（称为“T”位）为 1，表示队列已经到了末尾。还有一位，即 UHCI_PTR_QH，则表示所指向的是 uhci_qh 结构还是 uhci_td 结构。

我们已经建立起了一个以 uhci_qh 结构为头、通过物理地址链接的 uhci_td 结构队列，但这只是个游离于 USB 总线调度系统以外的队列，下一步要通过 uhci_insert_qh()把这个队列插入 USB 总线调度系统的队列。对于全速设备就是插在 uhci->skel_hs_control_qh 中，其代码见 drivers/usb/uhci.c：

==================== drivers/usb/uhci.c 331 348 ====================

[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_control()>uhci_insert_qh()]

```
static void uhci_insert_qh(struct uhci *uhci, struct uhci_qh *skelqh, struct uhci_qh *qh)
{
     unsigned long flags;

     spin_lock_irqsave(&uhci->framelist_lock, flags);

     /* Fix the linked list pointers */
     qh->nextqh = skelqh->nextqh;
     qh->prevqh = skelqh;
     if (skelqh->nextqh)
          skelqh->nextqh->prevqh = qh;
     skelqh->nextqh = qh;

     qh->link = skelqh->link;
     skelqh->link = virt_to_bus(qh) | UHCI_PTR_QH;

     spin_unlock_irqrestore(&uhci->framelist_lock, flags);
}
```

参数 skelqh 指向 USB 总线调度系统中的一个节点，这个节点本身也在一个队列中，而挂入这个节点的数据结构又是队列头，即代表着整个控制传输的交互请求队列。所以，节点 skelqh 所在的队列是

PDF-XCHANGE Click to buy NOW! www.docu-track.com

一个队列的队列，因此称之为“骨架”（skeleton）。USB 的调度系统中有两个骨架，一个是中断交互请求队列的骨架，另一个就是控制（以及成块）传输请求队列的骨架。另一个参数 qh，则指向为具体传输请求建立起来的交互请求队列。同样，uhci_qh 结构也与 USB 控制器硬件的操作有关；所以也是既有采用虚拟地址的链接指针 nextqh 和 prevqh，也有采用物理地址的链接指针 link。至于具体的队列操作，对于读者已经很简单了。这样，代表着本次传输的队列就链入了 USB 总线的调度系统中。

回到 uhci_submit_control()的代码中，对于全速设备，由于是横向执行，还要调用一个函数 uhci_inc_fsbr()，使 uhci 结构中的 skel_term_qh，即 skelqh[3]的指针 link，指向 uhci->skel_hs_control_qh。其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 563 580 ====================
[ioctl_scanner()>usb_control_msg()>usb_internal_control_msg()>usb_start_wait_urb()>usb_submit_urb()
>uhci_submit_urb()>uhci_submit_control()>uhci_inc_fsbr()]

```
static void uhci_inc_fsbr(struct uhci *uhci, struct urb *urb)
{
	unsigned long flags;
	struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;

	if (!urbp)
		return;

	spin_lock_irqsave(&uhci->framelist_lock, flags);

	if ((!(urb->transfer_flags & USB_NO_FSBR)) && (!urbp->fsbr)) {
		urbp->fsbr = 1;
		if (!uhci->fsbr++)
			uhci->skel_term_qh.link = virt_to_bus(&uhci->skel_hs_control_qh) | UHCI_PTR_QH;
	}

	spin_unlock_irqrestore(&uhci->framelist_lock, flags);
}
```

队列头 skel_term_qh 是整个调度系统的终点，当 USB 控制器执行到 skel_term_qh 时，当前框架的执行本来已经结束，可以等待下一个框架的开始了。可是，全速设备的控制交互队列是横向执行的，每次只从队列中执行一个交互便转入下一个队列，所以很可能一方面还有交互在等待执行，一方面 USB 控制器却已经空闲。既然如此，何不回过头来再执行全速设备的控制传输队列？在前面 alloc_uhci()的代码中（2229 行），uhci 结构中的成块传输队列头 skel_bulk_qh 的指针 link 已经设置成指向 uhci->skel_term_qh；就是说，在执行完优先级最低的成块传输队列以后，如果本框架中还有空闲时间，就转到“终结”队列头 skel_term_qh。如果这个节点是终点，那么本框架中剩余的时间就放弃了，USB 控制器在这段时间中“空转”等待下一个框架的开始。但是，如果让 skel_term_qh 回过头来链接到 skel_hs_control_qh，则可以在剩余的时间中再回过来执行控制传输队列，直至框架中剩下的时间已不足以完成一次交互时为止。当然，这只有在存在着采用横向执行的控制传输时才有意义。这里函数名（以及字段名）中的“fsbr”表示“全速设备带宽回收”（Full Speed Bandwidth Reclamation）。如果一个（全速设备的）传输请求没有明确表示不要 FSBR，就将其 fsbr 字段设成 1，并使 uhci 结构中的计数

PDF-XCHANGE Click to buy NOW! www.docu-track.com

器 fsbr 加 1，表示这个机制多了一个用户。如果原来这个计数器是 0 就将 skel_term_qh 反过来链接到 skel_hs_control_qh。

最后，通过 uhci_add_urb_list()把 urb 结构链入 uhci 结构的 urb_list。

至此，uhci_submit_control()的操作已经完成，CPU 经 uhci_submit_urb()返回到 usb_submit_urb()，并进而返回到 usb_start_wait_urb()的代码中，在那里进入睡眠（1019 行）。

如前所述，USB 控制器在每个时间框架中都从等时交互队列开始执行，然后是中断交互。这两种周期性传输（交互）所占的时间合在一起不超过一个框架的 90%，所以在执行完中断交互以后总是还有一些时间可以用来执行控制传输，然后才是成块传输。

具体执行时，USB 控制器因交互请求链入队列的方式而有不同的处理。我们在前面看到，控制（以及成块）交互描述块总是在其所属传输的队列中（指采用物理地址的队列，下同），都有个队列描述块（uhci_qh 结构）。我们称这样的队列为有头队列，并称有头队列内的交互请求是处于一个“队列上下文”中。USB 控制器在其内部维持一个执行堆栈，每碰到一个队列描述块就知道进入了一个新的队列上下文，离开该队列时就回到了此前的上下文。相比之下，等时交互的队列是没有队列头的，中断交互的队列也没有队列头，所以等时交互和中断交互都不在队列上下文中。对于在队列上下文中的交互（请求），USB 控制器在每执行完一个交互以后就自动改变队列头中的指针 element，使其“推进”指向队列中的下一个交互。从而，原来在队列最前面的交互，就是刚执行完毕的交互，就从队列中脱离了出来。所以，USB 控制器在队列上下文中总是执行由指针 element 所指的交互，称为队列的“顶部”。相比之下，如果 USB 控制器不在队列上下文中执行，则在完成一个交互以后并不“推进”指针（实际上也没有指针可以推进），而只是顺着链接指针执行队列中的下一个交互。因此，在这种情况下并不将已经执行的交互从队列中脱链。可想而知，这些交互请求还留在队列中，在下一轮循环中（1 秒钟以后）还会得到执行，这就保证了等时交互和中断交互的周期性。

这里要注意，“完成”当前交互、从而将交互请求中的 TD_CTRL_ACTIVE 标志位改成 0，并在队列上下文中推进指针，跟结束当前交互的执行（从而开始另一个变互的执行或开始空转），是不同的概念。所谓完成一个交互是指：完成了数据的传递并得到确认（如果是输出交互），或者连续发生超时或其它出错，使得继续重发已经失去意义。此时 USB 控制器将当前交互请求的 TD_CTRL_ACTIVE 标志位清成 0，并开始执行队列中的下一个交互请求或转入下一个队列，如果是在队列上下文则还要推进指针。而结束当前交互的执行，开始下一个交互请求或下一个队列的执行，则并不取决于当前交互的完成。例如，如果对方发回一个 NAK 表示暂时不能完成所要求的传输（如果发生在 SETUP 交互中则相当于一次超时错），或者因为得不对方的响应而超时，或者传递的过程中出了错（如 CRC 校验出错），则虽然当前的交互并未完成，但是对这个交互的本次执行也结束了。不同的是，此时不把它的 TD_CTRL_ACTIVE 标志位清 0，也不推进指针，如果是因出错而结束则还要把它的允许重发次数减 1。这样，当 USB 控制器下一次执行到同一队列、同一交互请求时又会再执行一次。

此外，在开始执行一个交互之前，USB 控制器先要判断是否能在本框架内结束这个交互，如果本框架中剩下的时间已经不够则宁可空转，等下一个框架再说。

控制传输的最后一个交互是状态交互。由于这个交互的 TD_CTRL_IOC 标志位为 1，USB 控制器在这个交互所在的框架结束时就向 CPU 发出一次中断请求。对于 UHCI 接口，USB 总线的中断服务程序为 uhci_interrupt()，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 2023 2085 ====================

```
static void uhci_interrupt(int irq, void *__uhci, struct pt_regs *regs)
{
        struct uhci *uhci = __uhci;
        unsigned int io_addr = uhci->io_addr;
        unsigned short status;
        unsigned long flags;
        struct list_head *tmp, *head;

        /*
         * Read the interrupt status, and write it back to clear the
         * interrupt cause
         */
        status = inw(io_addr + USBSTS);
        if (!status)  /* shared interrupt, not mine */
                return;
        outw(status, io_addr + USBSTS);

        if (status & ~(USBSTS_USBINT | USBSTS_ERROR)) {
                if (status & USBSTS_RD)
                        printk(KERN_INFO "uhci: resume detected, not implemented\n");
                if (status & USBSTS_HSE)
                        printk(KERN_ERR "uhci: host system error, PCI problems?\n");
                if (status & USBSTS_HCPE)
                        printk(KERN_ERR "uhci: host controller process error. something bad happened\n");
                if (status & USBSTS_HCH) {
                        printk(KERN_ERR "uhci: host controller halted. very bad\n");
                        /* FIXME: Reset the controller, fix the offending TD */
                }
        }

        uhci_free_pending_qhs(uhci);

        spin_lock(&uhci->urb_remove_lock);
        head = &uhci->urb_remove_list;
        tmp = head->next;
        while (tmp != head) {
                struct urb *urb = list_entry(tmp, struct urb, urb_list);

                tmp = tmp->next;

                list_del(&urb->urb_list);

                if (urb->complete)
                        urb->complete(urb);
        }
        spin_unlock(&uhci->urb_remove_lock);

        uhci_clear_next_interrupt(uhci);
```

```
2071
2072      /* Walk the list of pending TD's to see which ones completed */
2073      nested_lock(&uhci->urblist_lock, flags);
2074      head = &uhci->urb_list;
2075      tmp = head->next;
2076      while (tmp != head) {
2077           struct urb *urb = list_entry(tmp, struct urb, urb_list);
2078
2079           tmp = tmp->next;
2080
2081           /* Checks the status and does all of the magic necessary */
2082           uhci_transfer_result(urb);
2083      }
2084      nested_unlock(&uhci->urblist_lock, flags);
2085 }
```

首先读出 USB 控制器的中断状态寄存器，并将读出的内容写回该寄存器，这是典型的操作。我们先把 2053 行至 2069 行的代码暂搁一下，以后再回过来看。现在先看下面。

前面讲过，USB 控制器在执行中只要碰到一个交互描述块的 TD_CTRL_IOC 标志位为 1，就会在其所在的框架结束时向 CPU 发出中断请求。事实上，即使是一个不“活跃”、因而不需要执行的交互描述块也是一样，USB 控制器会跳过这个交互描述块，但还是会发出中断请求。这样，只要将 skel_term_td 的 TD_CTRL_IOC 标志位设成 1，就实际上每一毫秒就会中断一次。现在既然已经在中断服务程序中，就可以通过 uhci_clear_next_interrupt()把这个中断源暂时关闭（drivers/usb/uhci.c）。

==================== drivers/usb/uhci.c 135 143 ====================
[uhci_interrupt()>uhci_clear_next_interrupt()]

```
135  void uhci_clear_next_interrupt(struct uhci *uhci)
136  {
137       unsigned long flags;
138
139       spin_lock_irqsave(&uhci->framelist_lock, flags);
140       uhci->skel_term_td.status &= ~TD_CTRL_IOC;
141       spin_unlock_irqrestore(&uhci->framelist_lock, flags);
142  }
143
```

对于正常完成了操作的交互请求，USB 控制器已经自动将其从通过物理地址链接的队列中摘除，因而已经不再在 USB 总线的调度系统中。但是，这些数据结构还在相应 urb 结构（确切地说是 urb_priv 结构）中通过虚拟地址链接的队列中，而 urb 结构又链接在相应 USB 总线的 urb_list 中。现在就要扫描这个 urb 队列，通过 uhci_transfer_result()将其中已经完成了操作的传输请求找出来，并从队列中摘除，再“回叫”预先为这些传输安排好的善后操作。这个函数也是一个通用的函数，不仅仅是为控制交互安排的，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1379 1478 ====================
[uhci_interrupt()>uhci_transfer_result()]

```
1379 /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Return the result of a transfer
 *
 * Must be called with urblist_lock acquired
 */
static void uhci_transfer_result(struct urb *urb)
{
	struct usb_device *dev = urb->dev;
	struct urb *turb;
	int proceed = 0, is_ring = 0;
	int ret = -EINVAL;
	unsigned long flags;

	spin_lock_irqsave(&urb->lock, flags);

	switch (usb_pipetype(urb->pipe)) {
	case PIPE_CONTROL:
		ret = uhci_result_control(urb);
		break;
	case PIPE_INTERRUPT:
		ret = uhci_result_interrupt(urb);
		break;
	case PIPE_BULK:
		ret = uhci_result_bulk(urb);
		break;
	case PIPE_ISOCHRONOUS:
		ret = uhci_result_isochronous(urb);
		break;
	}

	urb->status = ret;

	spin_unlock_irqrestore(&urb->lock, flags);

	if (ret == -EINPROGRESS)
		return;

	switch (usb_pipetype(urb->pipe)) {
	case PIPE_CONTROL:
	case PIPE_BULK:
	case PIPE_ISOCHRONOUS:
		/* Release bandwidth for Interrupt or Isoc. transfers */
		/* Spinlock needed ? */
		if (urb->bandwidth)
			usb_release_bandwidth(urb->dev, urb, 1);
		uhci_unlink_generic(urb);
		break;
	case PIPE_INTERRUPT:
		/* Interrupts are an exception */
		if (urb->interval) {
```

```
            urb->complete(urb);
            uhci_reset_interrupt(urb);
            return;
        }

        /* Release bandwidth for Interrupt or Isoc. transfers */
        /* Spinlock needed ? */
        if (urb->bandwidth)
            usb_release_bandwidth(urb->dev, urb, 0);
        uhci_unlink_generic(urb);
        break;
    }

    if (urb->next) {
        turb = urb->next;
        do {
            if (turb->status != -EINPROGRESS) {
                proceed = 1;
                break;
            }

            turb = turb->next;
        } while (turb && turb != urb && turb != urb->next);

        if (turb == urb || turb == urb->next)
            is_ring = 1;
    }

    if (urb->complete && !proceed) {
        urb->complete(urb);
        if (!proceed && is_ring)
            uhci_submit_urb(urb);
    }

    if (proceed && urb->next) {
        turb = urb->next;
        do {
            if (turb->status != -EINPROGRESS &&
               uhci_submit_urb(turb) != 0)

            turb = turb->next;
        } while (turb && turb != urb->next);

        if (urb->complete)
            urb->complete(urb);
    }

    /* We decrement the usage count after we're done with everything */
    usb_dec_dev_use(dev);
```

```
}
```

参数 urb 指向链接在 urb_1ist 中的一个 urb 数据结构。显然，对于控制交互首先要对其调用 uhci_result_control()，这个函数的代码也在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 745 822 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_result_control()]

```
static int uhci_result_control(struct urb *urb)
{
	struct list_head *tmp, *head;
	struct urb_priv *urbp = urb->hcpriv;
	struct uhci_td *td;
	unsigned int status;
	int ret = 0;

	if (!urbp)
		return -EINVAL;

	head = &urbp->list;
	if (head->next == head)
		return -EINVAL;

	if (urbp->short_control_packet) {
		tmp = head->prev;
		goto status_phase;
	}

	tmp = head->next;
	td = list_entry(tmp, struct uhci_td, list);

	/* The first TD is the SETUP phase, check the status, but skip */
	/*  the count */
	status = uhci_status_bits(td->status);
	if (status & TD_CTRL_ACTIVE)
		return -EINPROGRESS;

	if (status)
		goto td_error;

	urb->actual_length = 0;

	/* The rest of the TD's (but the last) are data */
	tmp = tmp->next;
	while (tmp != head && tmp->next != head) {
		td = list_entry(tmp, struct uhci_td, list);

		tmp = tmp->next;

		if (urbp->fsbr_timeout && (td->status & TD_CTRL_IOC) &&
```

```
                      !(td->status & TD_CTRL_ACTIVE)) {
                        uhci_inc_fsbr(urb->dev->bus->hcpriv, urb);
                        urbp->fsbr_timeout = 0;
                        td->status &= ~TD_CTRL_IOC;
                }

                status = uhci_status_bits(td->status);
                if (status & TD_CTRL_ACTIVE)
                        return -EINPROGRESS;

                urb->actual_length += uhci_actual_length(td->status);

                if (status)
                        goto td_error;

                /* Check to see if we received a short packet */
                if (uhci_actual_length(td->status) < uhci_expected_length(td->info)) {
                        if (urb->transfer_flags & USB_DISABLE_SPD) {
                                ret = -EREMOTEIO;
                                goto err;
                        }

                        if (uhci_packetid(td->info) == USB_PID_IN)
                                return usb_control_retrigger_status(urb);
                        else
                                return 0;
                }
        }

status_phase:
        td = list_entry(tmp, struct uhci_td, list);

        /* Control status phase */
        status = uhci_status_bits(td->status);

#ifdef I_HAVE_BUGGY_APC_BACKUPS
```

==================== drivers/usb/uhci.c 830 857 ====================

```
#endif

        if (status & TD_CTRL_ACTIVE)
                return -EINPROGRESS;

        if (status)
                goto td_error;

        return 0;

td_error:
        ret = uhci_map_status(status, uhci_packetout(td->info));
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
842      if (ret == -EPIPE)
843           /* endpoint has stalled - mark it halted */
844           usb_endpoint_halt(urb->dev, uhci_endpoint(td->info),
845                     uhci_packetout(td->info));
846
847  err:
848      if (debug && ret != -EPIPE) {
849           /* Some debugging code */
850           dbg("uhci_result_control() failed with status %x", status);
851
852           /* Print the chain for debugging purposes */
853           uhci_show_urb_queue(urb);
854      }
855
856      return ret;
857  }
```

控制传输有三个阶段，其中第二个阶段有可能为空，所以其交互请求队列中至少应该有两个交互描述块，即使在特殊的情况下（见下）也至少还有最后的状态交互，如果队列为空就错了（757 行）。如果 urbp->short_control_packet 为 1，就说明在传输中曾发生接收到小于应有大小的信包，此时本次传输中只剩下最后一个交互，即状态交互，所以直接跳到状态交互（762 行），去检查其结果。

在正常的情况下，则扫描本次传输的交互请求队列，顺次检查各个交互描述块。如前所述，USB 总线上控制器会根据执行的结果改变其内容。首先是 SETUP 阶段的交互。如果一个交互的状态位 TD_CTRL_ACTIVE 仍旧是 1，那就说明尚未执行；或者执行失败了，但是失败的次数还在 3 次以下，需要再重新执行。所以返回-EINPROGRESS，表示还在进行中（772 行）。反之，如果 TD_CTRL_ACTIVE 已经变成了 0，那就说明至少 SETUP 阶段的交互已经完成了，此时若状态位中有任何一位为非 0 就表示出了错（775 行）。

下面，就要通过一个 while 循环检查数据阶段的各次交互了。786 行的条件语句目的在于优化。如果原来因为想要回收每个框架尾部的剩余时间而将最后一个队列头链接到了控制传输队列，可是实际上却在相当长一段时间内并没有起到作用，那就说明总线的负载太大了，此时一个由定时器触发的函数 rh_int_timer_do()就会将为了回收利用剩余时间而作的链接拆除（后面我们还要讲到这个过程），并把标志位 fsbr_timeout 设成 1。然后，当控制队列中的交互请求得到执行时，总线的负载可能已经轻下来，所以又可以通过 uhci_inc_fsbr()恢复 FSBR 链接了。

同样，如果交互描述块中的 TD_CTRL_ACTIVE 标志位为 1 就表示交互尚未完成，或者尚未彻底失败，所以返回-EINPROGRESS。

如果一个交互完成后的数据小于应有的长度（803 行），那就要看具体的情况。要是在提交该交互请求时把标志位 USB_DISABLE_SPD 设成 1（这里“SPD”是“short packet detect”，即“短信包检测”的意思），那就把该次交互视同失败，所以返回出错代码-EREMOTEIO。否则，如果是输入交互，就通过 usb_control_retrigger_status()重新调度本次传输，使 USB 总线的主控制器重新执行一遍最后的状态交互。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 859 901 ====================
[ubci_interrupt()>uhci_transfer_result()>uhci_result_control()>usb_control_retrigger_status()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
859 static int usb_control_retrigger_status(struct urb *urb)
860 {
861      struct list_head *tmp, *head;
862      struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
863      struct uhci *uhci = urb->dev->bus->hcpriv;
864
865      urbp->short_control_packet = 1;
866
867      /* Create a new QH to avoid pointer overwriting problems */
868      uhci_remove_qh(uhci, urbp->qh);
869
870      /* Delete all of the TD's except for the status TD at the end */
871      head = &urbp->list;
872      tmp = head->next;
873      while (tmp != head && tmp->next != head) {
874           struct uhci_td *td = list_entry(tmp, struct uhci_td, list);
875
876           tmp = tmp->next;
877
878           uhci_remove_td_from_urb(urb, td);
879
880           uhci_remove_td(uhci, td);
881
882           uhci_free_td(td);
883      }
884
885      urbp->qh = uhci_alloc_qh(urb->dev);
886      if (!urbp->qh) {
887           err("unable to allocate new QH for control retrigger");
888           return -ENOMEM;
889      }
890
891      /* One TD, who cares about Breadth first? */
892      uhci_insert_tds_in_qh(urbp->qh, urb, 0);
893
894      /* Low speed or small transfers gets a different queue and treatment */
895      if (urb->pipe & TD_CTRL_LS)
896           uhci_insert_qh(uhci, &uhci->skel_ls_control_qh, urbp->qh);
897      else
898           uhci_insert_qh(uhci, &uhci->skel_hs_control_qh, urbp->qh);
899
900      return -EINPROGRESS;
901 }
```

这里的 865 行把 urbp->short_control_packet 设成 1，并且另外创建一个只包括一个状态交互的队列，使目标设备仍能得到一个确认。这就是前面所说特殊情况的来历。

如果每个数据交互都正常完成，并且最后的状态交互也正常完成，uhci_result_control()便返回 0。回到 uhci_transfer_result()的代码中（drivers/usb/uhci.c，1396 行），如果从 uhci_result_control()返回

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的是-EINPROGRESS，即传输尚未完成，就让本次传输留在所调度的位置上不变而立即返回。否则就要将本次传输从队列中脱离出来，并释放所占用的“带宽”，即占用总线的时间。在 4 种不同的传输中，等时和中断两种是在调度时需要事先为其分配总线带宽的，这样才能保证同一框架中二者之和不超过 90%。如果预先分配了带宽，就一方面记录在（本次传输的）urb 结构中的 bandwidth 字段内，一方面也计入总线的 bus 结构中。对于控制交互，则虽然优先级别比成块传输为高，却并不需要预先分配带宽，因为对成块传输本来就无需保证其带宽。所以，对于控制交互实际上不会调用 usb_release_bandwidth()。但是，不管是控制、成块还是等时交互，都需要通过 uhci_unlink_generic()将其 urb 数据结构从各个队列中脱离出来。其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1480 1504 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_unlink_generic()]

```
static int uhci_unlink_generic(struct urb *urb)
{
	struct urb_priv *urbp = urb->hcpriv;
	struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;

	if (!urbp)
		return -EINVAL;

	uhci_dec_fsbr(uhci, urb);	/* Safe since it checks */

	uhci_remove_urb_list(uhci, urb);

	if (urbp->qh)
		/* The interrupt loop will reclaim the QH's */
		uhci_remove_qh(uhci, urbp->qh);

	if (!list_empty(&urbp->urb_queue_list))
		uhci_delete_queued_urb(uhci, urb);

	uhci_destroy_urb_priv(urb);

	urb->dev = NULL;

	return 0;
}
```

首先，由于本次传输的完成，对 FSBR，即对于框架末尾剩余时间的回收利用机制，就可能少了一个用户，所以通过 uhci_dec_fsbr()递减有关的计数，如果此后不再需要此种机制便拆除为此而建立的链接。然后通过 uhci_remove_urb_list()，使本次传输的 usb 数据结构脱离 uhci 结构中的队列。这两个函数的代码都很简单，我们就不看了。接着，如果是以传输（而不是交互）为单位调度的，就要通过 uhci_remove_qh()将 uhci_qh 数据结构从队列中脱链，因为 USB 控制器在执行的过程中会自动将交互描述块脱链，却不会自动将队列描述块脱链。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 350 383 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_unlink_generic()>uhci_remove_qh()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
static void uhci_remove_qh(struct uhci *uhci, struct uhci_qh *qh)
{
     unsigned long flags;
     int delayed;

     /* If the QH isn't queued, then we don't need to delay unlink it */
     delayed = (qh->prevqh || qh->nextqh);

     spin_lock_irqsave(&uhci->framelist_lock, flags);
     if (qh->prevqh) {
          qh->prevqh->nextqh = qh->nextqh;
          qh->prevqh->link = qh->link;
     }
     if (qh->nextqh)
          qh->nextqh->prevqh = qh->prevqh;
     qh->prevqh = qh->nextqh = NULL;
     qh->element = qh->link = UHCI_PTR_TERM;
     spin_unlock_irqrestore(&uhci->framelist_lock, flags);

     if (delayed) {
          spin_lock_irqsave(&uhci->qh_remove_lock, flags);

          /* Check to see if the remove list is empty */
          /* Set the IOC bit to force an interrupt so we can remove the QH */
          if (list_empty(&uhci->qh_remove_list))
               uhci_set_next_interrupt(uhci);

          /* Add it */
          list_add(&qh->remove_list, &uhci->qh_remove_list);

          spin_unlock_irqrestore(&uhci->qh_remove_lock, flags);
     } else
          uhci_free_qh(qh);
}
```

本来，将传输请求从队列中脱链以后就可以释放了。可是，USB 控制器的执行与 CPU 控制器的执行是互相独立的，所以有可能此时 USB 控制器正在这个队列中，所以要先把这个队列头转移到一个单独的队列 qh_remove_list 中，让它“冷却”一下（USB 控制器只能出而不能进），等下一次中断时再来释放。这就是我们在前面跳过的 uhci_free_pending_qhs()所作的处理（drivers/usb/uhci.c）。

==================== drivers/usb/uhci.c 2002 2021 ====================

[uhci_interrupt()>uhci_free_pending_qhs()]

```
void uhci_free_pending_qhs(struct uhci *uhci)
{
     struct list_head *tmp, *head;
     unsigned long flags;

     /* Free any pending QH's */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    spin_lock_irqsave(&uhci->qh_remove_lock, flags);
    head = &uhci->qh_remove_list;
    tmp = head->next;
    while (tmp != head) {
        struct uhci_qh *qh = list_entry(tmp, struct uhci_qh, remove_list);

        tmp = tmp->next;

        list_del(&qh->remove_list);

        uhci_free_qh(qh);
    }
    spin_unlock_irqrestore(&uhci->qh_remove_lock, flags);
}
```

回到 uhci_unlink_generic()的代码中，如果 urb_priv 结构中的队列头 urb_queue_list 非空，就说明当初提交传输请求时与其它（对同一设备同一端点的）传输合并了，所以要通过 uhci_delete_queued_urb()将其脱链。其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 438 478 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_unlink_generic()>uhci_delete_queued_urb()]

```
438 static void uhci_delete_queued_urb(struct uhci *uhci, struct urb *urb)
439 {
440     struct urb_priv *urbp, *nurbp;
441     unsigned long flags;
442
443     urbp = urb->hcpriv;
444
445     spin_lock_irqsave(&uhci_append_urb_lock, flags);
446
447     nurbp = list_entry(urbp->urb_queue_list.next, struct urb_priv,
448             urb_queue_list);
449
450     if (!urbp->queued) {
451         /* We're the head, so just insert the QH for the next URB */
452         uhci_insert_qh(uhci, &uhci->skel_bulk_qh, nurbp->qh);
453         nurbp->queued = 0;
454     } else {
455         struct urb_priv *purbp;
456         struct uhci_td *ptd;
457
458         /* We're somewhere in the middle (or end). A bit trickier */
459         /*  than the head scenario */
460         purbp = list_entry(urbp->urb_queue_list.prev, struct urb_priv,
461                 urb_queue_list);
462
463         ptd = list_entry(purbp->list.prev, struct uhci_td, list);
464         if (nurbp->queued)
465             /* Close the gap between the two */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                ptd->link = virt_to_bus(list_entry(nurbp->list.next,
                        struct uhci_td, list));
            else
                /* The next URB happens to be the beggining, so */
                /*  we're the last, end the chain */
                ptd->link = UHCI_PTR_TERM;

        }

        list_del(&urbp->urb_queue_list);

        spin_unlock_irqrestore(&uhci_append_urb_lock, flags);
}
```

最后通过 uhci_destroy_urb_priv()释放 urb_priv 数据结构，包括队列中的所有 uhci_td 结构，其代码也在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 523 561 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_unlink_generic()>uhci_destroy_urb_priv()]

```
static void uhci_destroy_urb_priv(struct urb *urb)
{
        struct list_head *tmp, *head;
        struct urb_priv *urbp;
        struct uhci *uhci;
        struct uhci_td *td;
        unsigned long flags;

        spin_lock_irqsave(&urb->lock, flags);

        urbp = (struct urb_priv *)urb->hcpriv;
        if (!urbp)
            goto unlock;

        if (!urb->dev || !urb->dev->bus || !urb->dev->bus->hcpriv)
            goto unlock;

        uhci = urb->dev->bus->hcpriv;

        head = &urbp->list;
        tmp = head->next;
        while (tmp != head) {
            td = list_entry(tmp, struct uhci_td, list);

            tmp = tmp->next;

            uhci_remove_td_from_urb(urb, td);

            uhci_remove_td(uhci, td);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            uhci_free_td(td);
        }

        urb->hcpriv = NULL;
        kmem_cache_free(uhci_up_cachep, urbp);

unlock:
        spin_unlock_irqrestore(&urb->lock, flags);
}
```

回到 uhci_transfer_result()的代码中（drivers/usb/uhci.c，1425 行），下面有一段特殊的代码（1442～1474 行）。这段代码主要是为等时传输而设计的，但也可以用于成块传输。

在传输的数据量比较大，又要求在时间上分布得比较均匀时，常常采用“双缓冲”甚至多缓冲的技术。例如对于音频信号，就可以设置两个缓冲区，使这两个缓冲区交替地用于接收数据和处理数据。这样可以使整个流量趋于平均，更为“流水线化”。具体到 USB 总线上的传输，可以为之准备下两个传输请求，即两个 urb 结构，并通过它们的指针 next 互相链接在一起。这样，当一个传输完成，因而要将其从调度系统中脱链时，便通过指针 next 找到其“配偶”，如果不在调度系统中就把它提交调度。这样，如果将这两个传输中的一个老是调度在前 1/2 秒，而另一个老是调度在后 1/2 秒，就可以使流量比只采用一个传输时平均了。至于一般的传输，则由于其指针 next 总是 0，因而代码中的 proceed 与 is_ring 总是 0，所以不受影响。

最后，只要为当前 urb 结构中的函数指针 complete 设置了善后操作，就要调用这个函数。前面在 usb_internal_control_msg()中把这个函数设置成 usb_api_blocking_completion()，所以现在就调用这个函数。其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 978 989 ====================

[uhci_interrupt()>uhci_transfer_result()>usb_api_blocking_completion()]

```
static void usb_api_blocking_completion(urb_t *urb)
{
        api_wrapper_data *awd = (api_wrapper_data *)urb->context;

        if (waitqueue_active(awd->wakeup))
            wake_up(awd->wakeup);
#if 0
        else
            dbg("(blocking_completion): waitqueue empty!");
            // even occurs if urb was unlinked by timeout...
#endif
}
```

对照一下前面 usb_start_wait_urb()的代码，就可以看出这里唤醒的正是前面调用了 usb_control_msg()的那个进程。这个进程当初是通过 schedule_timeout()睡眠等待（drivers/usb/usb.c，1019 行）的，所以被唤醒的原因可能有两个，即因为操作完成或者因为超时。如果是因超时而被唤醒，而传输实际上已经开始，那就再睡眠。最后，被唤醒的原因还是有两个，要么是传输完成了或者彻底失败了，要么是根本得不到执行而超时。如果是前者，那么 urb 结构已经从调度系统中脱离出来，而如

果是后者则 urb 结构仍旧链接在调度系统的队列中，所以先要把它脱离出来（1029 行）。然后，在释放了 urb 数据结构以后，从 usb_start_wait_urb()返回的 status 指示着传输的成败，而 actual_length 则说明已经传输的实际长度。至于从目标设备读回的数据，则放在预定的缓冲区中。这样，经过 usb_internal_control_msg()和 usb_control_msg()逐层返回到 ioctl_scanner()，本次传输操作已经完成。

我们在前面讲到，就扫描器的控制而言，其应用（驱动）进程可以由系统调用 ioctl()通过控制传输实现，也可以由一般的 write() / read()通过成块传输实现。但是，对于与扫描器间的数据传输，则只能由 write() / read()实现。

再看对扫描器的数据读 / 写，这是通过成块传输完成的。对于扫描器，读操作显然更为重要，所以我们只看系统调用 read()的实现 read_scanner()。其代码在 drivers/usb/scanner.c 中：

==================== drivers/usb/scanner.c 518 624 ====================

```
518  static ssize_t
519  read_scanner(struct file * file, char * buffer,
520               size_t count, loff_t *ppos)
521  {
522       struct scn_usb_data *scn;
523       struct usb_device *dev;
524
525       ssize_t bytes_read;     /* Overall count of bytes_read */
526       ssize_t ret;
527
528       kdev_t scn_minor;
529
530       int partial;       /* Number of bytes successfully read */
531       int this_read;          /* Max number of bytes to read */
532       int result;
533       int rd_expire = RD_EXPIRE;
534
535       char *ibuf;
536
537       scn = file->private_data;
538
539       scn_minor = scn->scn_minor;
540
541       ibuf = scn->ibuf;
542
543       dev = scn->scn_dev;
544
545       bytes_read = 0;
546       ret = 0;
547
548       file->f_dentry->d_inode->i_atime = CURRENT_TIME; /* Update the
549                                                       atime of
550                                                       the device
551                                                       node */
552       down(&(scn->gen_lock));
553
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	while (count > 0) {
		if (signal_pending(current)) {
			ret = -EINTR;
			break;
		}

		this_read = (count >= IBUF_SIZE) ? IBUF_SIZE : count;

		result = usb_bulk_msg(dev, usb_rcvbulkpipe(dev, scn->bulk_in_ep), ibuf, this_read, &partial, RD_NAK_TIMEOUT);
		dbg("read stats(%d): result:%d this_read:%d partial:%d count:%d", scn_minor, result, this_read, partial, count);

/*
 * Scanners are sometimes inheriently slow since they are mechanical
 * in nature.  USB bulk reads tend to timeout while the scanner is
 * positioning, resetting, warming up the lamp, etc if the timeout is
 * set too low.  A very long timeout parameter for bulk reads was used
 * to overcome this limitation, but this sometimes resulted in folks
 * having to wait for the timeout to expire after pressing Ctrl-C from
 * an application. The user was sometimes left with the impression
 * that something had hung or crashed when in fact the USB read was
 * just waiting on data.  So, the below code retains the same long
 * timeout period, but splits it up into smaller parts so that
 * Ctrl-C's are acted upon in a reasonable amount of time.
 */

		if (result == USB_ST_TIMEOUT && !partial) { /* Timeout
							     and no
							     data */
			if (--rd_expire <= 0) {
				warn("read_scanner(%d): excessive NAK's received", scn_minor);
				ret = -ETIME;
				break;
			} else {
				interruptible_sleep_on_timeout(&scn->rd_wait_q, RD_NAK_TIMEOUT);
				continue;
			}
		} else if ((result < 0) && (result != USB_ST_DATAUNDERRUN)) {
			warn("read_scanner(%d): funky result:%d. Please notify the maintainer.", scn_minor, (int)result);
			ret = -EIO;
			break;
		}

#ifdef RD_DATA_DUMP
		if (partial) {
			unsigned char cnt, cnt_max;
			cnt_max = (partial > 24) ? 24 : partial;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
600                 printk(KERN_DEBUG "dump(%d): ", scn_minor);
601                 for (cnt=0; cnt < cnt_max; cnt++) {
602                      printk("%X ", ibuf[cnt]);
603                 }
604                 printk("\n");
605            }
606  #endif
607
608            if (partial) { /* Data returned */
609                 if (copy_to_user(buffer, ibuf, partial)) {
610                      ret = -EFAULT;
611                      break;
612                 }
613                 count -= this_read; /* Compensate for short reads */
614                 bytes_read += partial; /* Keep tally of what actually was read */
615                 buffer += partial;
616            } else {
617                 ret = 0;
618                 break;
619            }
620       }
621       up(&(scn->gen_lock));
622
623       return ret ? ret : bytes_read;
624  }
```

我们先把面向扫描器本身的代码留给读者，在这里把注意力集中在对成块传输的调度 usb_bulk_msg()上。由于我们已经详细阅读了有关控制传输的代码，下面只需说明二者的不同之处。

如前所述，每个 USB 设备的 0 号端点总是用于控制传输的，并且是双向的端点。除此之外，则设备中的每个“接口”，即逻辑功能都有一组端点，不同类型的传输要使用不同的端点，并且每个端点都是单向的。这些端点的号码都在相应 USB 设备的枚举阶段通过控制传输取得。取得了一个端点号以后，主机与这个端点之间的就形成了一个逻辑上的“管道”。与控制交互不同，成块传输只有一个阶段，即数据阶段，可以包括一次或多次数据交互。每次交互也是由三个信包构成，第一个总是由主控制器发出的 token 信包，其中包含着对方的地址和端点号，以及交互的类型，即 PID。对于成块传输的 PID，include/linux/usb.h 中定义了两个宏操作：

==================== include/linux/usb.h 749 750 ====================

```
749  #define usb_sndbulkpipe(dev,endpoint)((PIPE_BULK << 30) | __create_pipe(dev,endpoint))
750  #define usb_rcvbulkpipe(dev,endpoint)((PIPE_BULK << 30) | __create_pipe(dev,endpoint) | USB_DIR_IN)
```

成块传输是通过 usb_bulk_msg()完成的，这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 1111 1147 ====================

[read_scanner()>usb_bulk_msg()]

```
1111 /**
1112  *   usb_bulk_msg - Builds a bulk urb, sends it off and waits for completion
1113  *   @usb_dev: pointer to the usb device to send the message to
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 *   @pipe: endpoint "pipe" to send the message to
 *   @data: pointer to the data to send
 *   @len: length in bytes of the data to send
 *   @actual_length: pointer to a location to put the actual length transfered in bytes
 *   @timeout: time to wait for the message to complete before timing out (if 0 the wait is forever)
 *
 *   This function sends a simple bulk message to a specified endpoint
 *   and waits for the message to complete, or timeout.
 *
 *   If successful, it returns 0, othwise a negative error number.
 *   The number of actual bytes transferred will be plaed in the
 *   actual_timeout paramater.
 *
 *   Don't use this function from within an interrupt context, like a
 *   bottom half handler.  If you need a asyncronous message, or need to
 *   send a message from within interrupt context, use usb_submit_urb()
 */
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
              void *data, int len, int *actual_length, int timeout)
{
     urb_t *urb;

     if (len < 0)
          return -EINVAL;

     urb=usb_alloc_urb(0);
     if (!urb)
          return -ENOMEM;

     FILL_BULK_URB(urb,usb_dev,pipe,(unsigned char*)data,len,   /* build urb */
               (usb_complete_t)usb_api_blocking_completion,0);

     return usb_start_wait_urb(urb,timeout,actual_length);
}
```

对照前面由 usb_control_msg()调用的 usb_internal_control_msg()，就可以看出二者几乎一样，只是对成块传输请求的 urb 数据结构是通过 FILL_BULK_URB（而不是 FILL_CONTROL_URB）完成的。这个宏操作定义于 include/linux/usb.h：

==================== include/linux/usb.h 480 489 ====================

```
480  #define FILL_BULK_URB(a,aa,b,c,d,e,f) \
481     do {\
482      spin_lock_init(&(a)->lock);\
483      (a)->dev=aa;\
484      (a)->pipe=b;\
485      (a)->transfer_buffer=c;\
486      (a)->transfer_buffer_length=d;\
487      (a)->complete=e;\
488      (a)->context=f;\
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
489     } while (0)
```

读者不妨比较一下，看看二者有何不同，以及为什么会有不同。

从这以后，一直要到 uhci_submit_urb()中，成块传输与控制传输才又有不同。我们在这里只列出这个函数中用于成块传输的片段（drivers/usb/uhci.c）：

==================== drivers/usb/uhci.c 1342 1344 ====================

[read_scanner()>usb_bulk_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()]

```
1342     case PIPE_BULK:
1343          ret = uhci_submit_bulk(urb, u);
1344          break;
```

显然，控制传输请求的提交由 uhci_submit_bulk()完成，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1039 1115 ====================

[read_scanner()>usb_bulk_msg()>usb_start_wait_urb()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_bulk()]

```
1039 static int uhci_submit_bulk(struct urb *urb, struct urb *eurb)
1040 {
1041     struct uhci_td *td;
1042     struct uhci_qh *qh;
1043     unsigned long destination, status;
1044     struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1045     int maxsze = usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe));
1046     int len = urb->transfer_buffer_length;
1047     unsigned char *data = urb->transfer_buffer;
1048     struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
1049
1050     if (len < 0)
1051          return -EINVAL;
1052
1053     /* Can't have low speed bulk transfers */
1054     if (urb->pipe & TD_CTRL_LS)
1055          return -EINVAL;
1056
1057     /* The "pipe" thing contains the destination in bits 8--18 */
1058     destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
1059
1060     /* 3 errors */
1061     status = TD_CTRL_ACTIVE | (3 << TD_CTRL_C_ERR_SHIFT);
1062
1063     if (!(urb->transfer_flags & USB_DISABLE_SPD))
1064          status |= TD_CTRL_SPD;
1065
1066     /*
1067      * Build the DATA TD's
1068      */
1069     do { /* Allow zero length packets */
1070          int pktsze = len;
1071
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1072          if (pktsze > maxsze)
1073              pktsze = maxsze;
1074
1075          td = uhci_alloc_td(urb->dev);
1076          if (!td)
1077              return -ENOMEM;
1078
1079          uhci_add_td_to_urb(urb, td);
1080          uhci_fill_td(td, status, destination | ((pktsze - 1) << 21) |
1081              (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1082               usb_pipeout(urb->pipe)) << TD_TOKEN_TOGGLE),
1083              virt_to_bus(data));
1084
1085          data += pktsze;
1086          len -= maxsze;
1087
1088          if (len <= 0)
1089              td->status |= TD_CTRL_IOC;
1090
1091          usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe),
1092              usb_pipeout(urb->pipe));
1093      } while (len > 0);
1094
1095      qh = uhci_alloc_qh(urb->dev);
1096      if (!qh)
1097          return -ENOMEM;
1098
1099      urbp->qh = qh;
1100
1101      /* Always assume depth first */
1102      uhci_insert_tds_in_qh(qh, urb, 1);
1103
1104      if (urb->transfer_flags & USB_QUEUE_BULK && eurb) {
1105          urbp->queued = 1;
1106          uhci_append_queued_urb(uhci, eurb, urb);
1107      } else
1108          uhci_insert_qh(uhci, &uhci->skel_bulk_qh, qh);
1109
1110      uhci_add_urb_list(uhci, urb);
1111
1112      uhci_inc_fsbr(uhci, urb);
1113
1114      return -EINPROGRESS;
1115 }
```

读过 uhci_submit_control()的代码的读者对此自然不会有困难，我们只指出几点。第一，成块传输只能用于全速设备，（试想，如果是低速设备而又要“成块”传输，怎么能在一个框架，即 1 毫秒的时间里完成？）所以如果是低速设备（1054 行）就立即返回出错代码-EINVAL。第二，成块传输只有一

PDF-XCHANGE Click to buy NOW! www.docu-track.com

个阶段、一种交可，即数据交互，所以不像控制传输那样还有 SETUP 交互和状态交互。第三，对成块传输队列的执行总是横向，即宽度优先的，所以调用 uhci_insert_tds_in_qh()时的最后一个参数 breadth 总是 1（1101 行的注释中说“ Always assume depth first”正好是说反了）。还有，对成块传输允许将针对同一对象的传输合并（1106 行），所以 uhci_submit_control()多一个参数 eurb，非 0 时指向已经存在而尚未完成、针对同一对象的成块传输请求。

同样，请求传输的进程也在 usb_start_wait_urb()中睡眠等待，而 USB 控制器在最后一个交互所在的框架结束时会向 CPU 发出中断请求，USB 总线中断服务程序的入口也同样是 uhci_interrupt()。不同的是：此时在 uhci_transfer_result()中调用的是 uhci_result_bulk()，而不是 uhci_result_control()。下面是 uhci_transfer_result()中的一个片段（drivers/usb/uhci.c）：

==================== drivers/usb/uhci.c 1401 1403 ====================

[uhci_interrupt()>uhci_transfer_result()]

```
    . . . . . .
    case PIPE_BULK:
        ret = uhci_result_bulk(urb);
        break;
    . . . . . .
```

除此以外，就都与控制传输相同了。至于 uhci_result_bulk()，则在 drivers/usb/uhci.c 中定义为 uhci_result_interrupt()：

==================== drivers/usb/uhci.c 1117 1118 ====================

```
/* We can use the result interrupt since they're identical */
#define uhci_result_bulk uhci_result_interrupt
```

实际上，成块传输与中断传输所传递的都是数据，都是可靠传递（如出错就要重发），只是数据量和（调度）启动的方式不同，所以传输结束时可以由同一个函数加以处理。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 940 1012 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_result_interrupt()]

```
static int uhci_result_interrupt(struct urb *urb)
{
    struct list_head *tmp, *head;
    struct urb_priv *urbp = urb->hcpriv;
    struct uhci_td *td;
    unsigned int status;
    int ret = 0;

    if (!urbp)
        return -EINVAL;

    urb->actual_length = 0;

    head = &urbp->list;
    tmp = head->next;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    while (tmp != head) {
        td = list_entry(tmp, struct uhci_td, list);

        tmp = tmp->next;

        if (urbp->fsbr_timeout && (td->status & TD_CTRL_IOC) &&
           !(td->status & TD_CTRL_ACTIVE)) {
            uhci_inc_fsbr(urb->dev->bus->hcpriv, urb);
            urbp->fsbr_timeout = 0;
            td->status &= ~TD_CTRL_IOC;
        }

        status = uhci_status_bits(td->status);
        if (status & TD_CTRL_ACTIVE)
            return -EINPROGRESS;

        urb->actual_length += uhci_actual_length(td->status);

        if (status)
            goto td_error;

        if (uhci_actual_length(td->status) < uhci_expected_length(td->info)) {
            usb_settoggle(urb->dev, uhci_endpoint(td->info),
                uhci_packetout(td->info),
                uhci_toggle(td->info) ^ 1);

            if (urb->transfer_flags & USB_DISABLE_SPD) {
                ret = -EREMOTEIO;
                goto err;
            } else
                return 0;
        }
    }

    return 0;

td_error:
    ret = uhci_map_status(status, uhci_packetout(td->info));
    if (ret == -EPIPE)
        /* endpoint has stalled - mark it halted */
        usb_endpoint_halt(urb->dev, uhci_endpoint(td->info),
                uhci_packetout(td->info));

err:
    if (debug && ret != -EPIPE) {
        /* Some debugging code */
        dbg("uhci_result_interrupt/bulk() failed with status %x",
            status);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        /* Print the chain for debugging purposes */
        if (urbp->qh)
            uhci_show_urb_queue(urb);
        else
            uhci_show_td(td);
    }

    return ret;
}
```

同样，认真读过 uhci_result_control()的读者也不应对这段代码感到困难。

读者自然会问：扫描器是速度比较慢的设备，怎么能知道在什么时候去读就有数据可读呢？确实是这样。扫描器（以及其它 USB 设备）在接收到一个 IN 命令（token 信包）时，如果没有数据可以发送就会发送一个 NAK 作为应答。而 USB 控制器，则在数据交互（以及中断交互）中接收到 NAK 时不将交互请求的 TD_CTRL_ACTIVE 位设置成 0，也不推进（本次传输的）队列头中的 element 指针，就好像不曾启动本次交互一样。当然，最后很可能会使相应的传输因超时而失败。所以，从本质上讲，一次 IN 交互（以及 OUT 交互）实际上也是一次查询。那么，为什么不像一般设备驱动那样，让设备在有数据可发送时就主动向主机发出一个中断请求呢？前面讲过，USB 总线的所谓“中断交互”在本质上也是查询，只不过中断交互是周期性的，其执行是有保证的，而成块交互既非周期性也无保证而已。当然，USB 设备只能被动地等待受查询这么一种安排会在一定程度上降低效率，但这是 USB 总线的设计人员在各方面权衡折中以后作出的选择。如果考虑到 USB 总线上相当大比例的流量将是等时传输（不需要查询），再考虑到总线上设备的个数有限（127）以及总线的速度，并且 USB 控制器本身就带有微处理器，这样的选择应该说是合理的。注意上面所说的超时是指整个传输（而不是某次交互）的超时，在初始化根集中器时设置了一个定时器，定时地通过一个函数 rh_int_timer_do()扫描 USB 总线的 urb_list 队列，将超时的传输请求从队列中摘除。

回到 read_scanner()的代码中。既然成块传输很可能会因超时而失败，就要把这一点考虑进去。怎么办呢？很简单，只要传输超时，就醒了再睡，过一会儿再来试试（见代码中的 579、587 和 588 行）。也就是说，在较高的层次上也实现定时查询。这样的查询当然不能无限制地进行下去，所以代码中安排了一个计数器 rd_expire。

在扫描器的驱动程序中并未用到中断传输，但是中断传输在对 USB 集中器的操作（从而 USB 设备的热插入）中扮演着重要的角色，所以也应该看一下。我们在前面 usb_hub_configure()的代码中看到，要调度一个（周期性的）中断传输时，需要为其分配和设置好一个 urb 数据结构，并通过 usb_submit_urb()提交请求（drivers/usb/hub.c，214～223 行）。

同样，对于采用 UHCI 界面的主控制器，usb_submit_urb()会调用 uhci_submit_urb()完成对传输的调度。也正是在这里，对中断传输的调度有了一些与此前不同的操作。下面是 uhci_submit_urb()中的一个片段（drivers/usb/uhci.c）：

==================== drivers/usb/uhci.c 1329 1341 ====================
[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()]

```
    . . . . . .
    case PIPE_INTERRUPT:
        if (urb->bandwidth == 0) {  /* not yet checked/allocated */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            bustime = usb_check_bandwidth(urb->dev, urb);
            if (bustime < 0)
                ret = bustime;
            else {
                ret = uhci_submit_interrupt(urb);
                if (ret == -EINPROGRESS)
                    usb_claim_bandwidth(urb->dev, urb, bustime, 0);
            }
        } else        /* bandwidth is already set */
            ret = uhci_submit_interrupt(urb);
        break;
  . . . . . .
```

中断传输的执行必须是有保证的，所以要预先通过 usb_check_bandwidth()为之分配“带宽”，即占用总线的时间。这个函数是中断传输和等时传输公用的，其代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 256 305 ====================

[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()>usb_check_bandwidth()]

```
/*
 * usb_check_bandwidth():
 *
 * old_alloc is from host_controller->bandwidth_allocated in microseconds;
 * bustime is from calc_bus_time(), but converted to microseconds.
 *
 * returns <bustime in us> if successful,
 * or USB_ST_BANDWIDTH_ERROR if bandwidth request fails.
 *
 * FIXME:
 * This initial implementation does not use Endpoint.bInterval
 * in managing bandwidth allocation.
 * It probably needs to be expanded to use Endpoint.bInterval.
 * This can be done as a later enhancement (correction).
 * This will also probably require some kind of
 * frame allocation tracking...meaning, for example,
 * that if multiple drivers request interrupts every 10 USB frames,
 * they don't all have to be allocated at
 * frame numbers N, N+10, N+20, etc.  Some of them could be at
 * N+11, N+21, N+31, etc., and others at
 * N+12, N+22, N+32, etc.
 * However, this first cut at USB bandwidth allocation does not
 * contain any frame allocation tracking.
 */
int usb_check_bandwidth (struct usb_device *dev, struct urb *urb)
{
    int        new_alloc;
    int        old_alloc = dev->bus->bandwidth_allocated;
    unsigned int  pipe = urb->pipe;
    long       bustime;

```

```
	bustime = usb_calc_bus_time (usb_pipeslow(pipe), usb_pipein(pipe),
			usb_pipeisoc(pipe), usb_maxpacket(dev, pipe, usb_pipeout(pipe)));
	if (usb_pipeisoc(pipe))
		bustime = NS_TO_US(bustime) / urb->number_of_packets;
	else
		bustime = NS_TO_US(bustime);

	new_alloc = old_alloc + (int)bustime;
		/* what new total allocated bus time would be */

	if (new_alloc > FRAME_TIME_MAX_USECS_ALLOC)
		dbg("usb-check-bandwidth %sFAILED: was %u, would be %u, bustime = %ld us",
			usb_bandwidth_option ? "" : "would have ",
			old_alloc, new_alloc, bustime);

	if (!usb_bandwidth_option)  /* don't enforce it */
		return (bustime);
	return (new_alloc <= FRAME_TIME_MAX_USECS_ALLOC) ? bustime : USB_ST_BANDWIDTH_ERROR;
}
```

首先，根据目标设备是否为低速设备、传输的方向和类型以及目标设备允许的信包大小（这就是本次传输实际传递的数据量），通过 usb_calc_bus_time()计算出需要的带宽。这个函数的代码在 drivers/usb/usb.c 中：

==================== drivers/usb/usb.c 219 254 ====================

[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()>usb_check_bandwidth()>usb_calc_bus_time()]

```
/*
 * usb_calc_bus_time:
 *
 * returns (approximate) USB bus time in nanoseconds for a USB transaction.
 */
static long usb_calc_bus_time (int low_speed, int input_dir, int isoc, int bytecount)
{
	unsigned long tmp;

	if (low_speed)		/* no isoc. here */
	{
		if (input_dir)
		{
			tmp = (67667L * (31L + 10L * BitTime (bytecount))) / 1000L;
			return (64060L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
		}
		else
		{
			tmp = (66700L * (31L + 10L * BitTime (bytecount))) / 1000L;
			return (64107L + (2 * BW_HUB_LS_SETUP) + BW_HOST_DELAY + tmp);
		}
	}

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    /* for full-speed: */

    if (!isoc)         /* Input or Output */
    {
        tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
        return (9107L + BW_HOST_DELAY + tmp);
    } /* end not Isoc */

    /* for isoc: */

    tmp = (8354L * (31L + 10L * BitTime (bytecount))) / 1000L;
    return (((input_dir) ? 7268L : 6265L) + BW_HOST_DELAY + tmp);
}
```

具体的计算涉及 USB 总线的物理性质，我们在这里从略，有必趣或需要的读者可参阅 USB 总线 1.1 版的规格书。计算的结果是以毫微秒为单位的占用总线时间。

回到 usb_check_bandwidth()中，又通过 NS_TO_US 将毫微秒换算成微秒。如果是等时传输则还要除以需要传递的数据信包数量，因为每个数据信包构成一次交互，即一个调度单位。属于同一传输的不同交互一般依次分布在不同的框架中，而不在同一框架中完成。至于中断传输，则只有一个交互，一个信包。连同原来已经分配的带宽一起，总的带宽不得超过定义于 include/linux/usb.h 的常数 FRAME_TIME_MAX_USECS_ALLOC：

==================== include/linux/usb.h 795 796 ====================

```
#define FRAME_TIME_USECS    1000L
#define FRAME_TIME_MAX_USECS_ALLOC   (90L * FRAME_TIME_USECS / 100L)
```

就是说，不得超过 1000 微秒的 90％。

为了对框架的容量有个大概的印象，我们在此不妨作一个粗略的估算。对于全速设备，USB 总线的理论带宽是 12Mb，即 1.5MB，除以 1024，则每个框架的带宽大约是 1.5KB，实际上当然达不到这么高。如果每个等时交互的信包大小为 1023 字节，则实际上在每个框架中只能调度一个等时交互；如果信包大小改成 512 宁节，那么也只能调度两个等时交互；余可类推。

回到 uhci_submit_urb()的代码中，如果可以分配所需的带宽，就可以进一步通过 uhci_submit_interrupt()提交传输请求了。其代码见 drivers/usb/uhci.c。

==================== drivers/usb/uhci.c 906 938 ====================

[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_interrupt()]

```
static int uhci_submit_interrupt(struct urb *urb)
{
    struct uhci_td *td;
    unsigned long destination, status;
    struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;

    if (urb->transfer_buffer_length > usb_maxpacket(urb->dev, urb->pipe, usb_pipeout(urb->pipe)))
        return -EINVAL;

    /* The "pipe" thing contains the destination in bits 8--18 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);

    status = (urb->pipe & TD_CTRL_LS) | TD_CTRL_ACTIVE | TD_CTRL_IOC;

    td = uhci_alloc_td(urb->dev);
    if (!td)
         return -ENOMEM;

    destination |= (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe), usb_pipeout(urb->pipe)) << TD_TOKEN_TOGGLE);
    destination |= ((urb->transfer_buffer_length - 1) << 21);

    usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe), usb_pipeout(urb->pipe));

    uhci_add_td_to_urb(urb, td);
    uhci_fill_td(td, status, destination,
         virt_to_bus(urb->transfer_buffer));

    uhci_insert_td(uhci, &uhci->skeltd[__interval_to_skel(urb->interval)], td);

    uhci_add_urb_list(uhci, urb);

    return -EINPROGRESS;
}
```

这个函数与前面的 uhci_submit_bulk()很相似，只是成块传输可以有多个数据信包，而中断传输只有一个信包。不过，这二者还有个重要的区别，就是成块传输以传输为调度单位，插入调度队列的是一个代表着成块传输请求的队列（队列头和若干交互请求）；而中断传输则直接把交互请求插入所有中断交互的队列。那么，具体插入到什么位置上呢？以前讲过，数组 skeltd[]是中断交互队列的“骨架”，把交互请求插入到这个骨架的哪一点上就决定了中断传输的周期。所以，代码中根据目标设备的中断传输周期通过__interval_to_skel()计算出插入点的下标，其代码见 drivers/usb/uhci.h。

==================== drivers/usb/uhci.h 253 289 ====================

[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_interrupt()>__interval_to_skel()]

```
/*
 * Search tree for determining where <interval> fits in the
 * skelqh[] skeleton.
 *
 * An interrupt request should be placed into the slowest skelqh[]
 * which meets the interval/period/frequency requirement.
 * An interrupt request is allowed to be faster than <interval> but not slower.
 *
 * For a given <interval>, this function returns the appropriate/matching
 * skelqh[] index value.
 *
 * NOTE: For UHCI, we don't really need int256_qh since the maximum interval
 * is 255 ms.  However, we do need an int1_qh since 1 is a valid interval
 * and we should meet that frequency when requested to do so.
```

```
 * This will require some change(s) to the UHCI skeleton.
 */
static inline int __interval_to_skel(int interval)
{
     if (interval < 16) {
          if (interval < 4) {
               if (interval < 2)
                    return 0;/* int1 for 0-1 ms */
               return 1;     /* int2 for 2-3 ms */
          }
          if (interval < 8)
               return 2;     /* int4 for 4-7 ms */
          return 3;          /* int8 for 8-15 ms */
     }
     if (interval < 64) {
          if (interval < 32)
               return 4;     /* int16 for 16-31 ms */
          return 5;          /* int32 for 32-63 ms */
     }
     if (interval < 128)
          return 6;          /* int64 for 64-127 ms */
     return 7;               /* int128 for 128-255 ms (Max.) */
}
```

将中断交互请求插入调度队列以后，还要通过 usb_claim_bandwidth()记下一笔账，以免把同一带宽又重复分配给其它传输。其代码见 drivers/usb/usb.c。

==================== drivers/usb/usb.c 307 322 ====================

[usb_hub_configure()>usb_submit_urb()>uhci_submit_urb()>uhci_submit_interrupt()>usb_claim_bandwidth()]

```
void usb_claim_bandwidth (struct usb_device *dev, struct urb *urb, int bustime, int isoc)
{
     dev->bus->bandwidth_allocated += bustime;
     if (isoc)
          dev->bus->bandwidth_isoc_reqs++;
     else
          dev->bus->bandwidth_int_reqs++;
     urb->bandwidth = bustime;

#ifdef USB_BANDWIDTH_MESSAGES
     dbg("bandwidth alloc increased by %d to %d for %d requesters",
          bustime,
          dev->bus->bandwidth_allocated,
          dev->bus->bandwidth_int_reqs + dev->bus->bandwidth_isoc_reqs);
#endif
}
```

传输完成以后，在中断服务程序中会调用 uhci_transfer_result()。前面我们已经看到，uhci_result_bulk()实际上也是 uhci_resuit_interrupt()。

对于 USB 控制器而言，中断交互队列并没有队列头，所以对中断交互的执行不在“队列上下文”

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

中。这样，USB 控制器在执行完一个中断交互时就无从推进队列头中的指针（根本就不存在队列头）；从而不会将中断交互从（物理地址）队列中脱链，下一次轮到同一个框架时还会执行这个交互请求。USB 控制器的这种特殊设计，与软件上的安排结合在一起，就保证了中断交互（以及等时交互）的周期性。相比之下，控制交互与成块交互在执行完成以后就从（物理地址）队列中脱链了。

对中断交互，在 uhci_transfer_result()中执行了 uhci_result_interrupt()以后还有一些特殊的操作，我们再列出有关的代码片断，以便阅读（drivers/usb/uhci.c）：

==================== drivers/usb/uhci.c 1416 1419 ====================

[uhci_interrupt()>uhci_transfer_result()]

```
1416      switch (usb_pipetype(urb->pipe)) {
1417      case PIPE_CONTROL:
1418      case PIPE_BULK:
1419      case PIPE_ISOCHRONOUS:
     . . . . . .
```

==================== drivers/usb/uhci.c 1425 1440 ====================

```
1425           break;
1426      case PIPE_INTERRUPT:
1427           /* Interrupts are an exception */
1428           if (urb->interval) {
1429                urb->complete(urb);
1430                uhci_reset_interrupt(urb);
1431                return;
1432           }
1433
1434           /* Release bandwidth for Interrupt or Isoc. transfers */
1435           /* Spinlock needed ? */
1436           if (urb->bandwidth)
1437                usb_release_bandwidth(urb->dev, urb, 0);
1438           uhci_unlink_generic(urb);
1439           break;
1440      }
```

中断传输是周期性的，usb 结构中的 interval 字段决定了它的周期。显然，这个周期不应该是 0，所以可以用 0 来表示特殊的意义。事实上，当 interval 为 0 时就表示相应的传输是一次性的（而不是周期性的）操作。所以，代码中（1436～1439 行）当 interval 为 0 时就通过 usb_release_bandwidth()释放带宽、并且通过 uhci_unlink_generic()释放有关的数据结构。与其它三种传输在此时的操作（1422～1424 行）相比较，就可以看出完全是一样的。但是，一次性的中断传输毕竟是特殊情况，实际上 interval 总是非 0，否则中断传输就失去了存在的意义。当 interval 非 0 时，一方面先调用事先设置的 complete 函数，唤醒正在睡眠等待的进程，或者向其发送一个信号；另一方面则调用一个函数 uhci_reset_interrupt()，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1425 1440 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_reset_interrupt()]

```
1014 static void uhci_reset_interrupt(struct urb *urb)
1015 {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1016      struct list_head *tmp;
1017      struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
1018      struct uhci_td *td;
1019
1020      if (!urbp)
1021           return;
1022
1023      tmp = urbp->list.next;
1024      td = list_entry(tmp, struct uhci_td, list);
1025      if (!td)
1026           return;
1027
1028      td->status = (td->status & 0x2F000000) | TD_CTRL_ACTIVE | TD_CTRL_IOC;
1029      td->info &= ~(1 << TD_TOKEN_TOGGLE);
1030      td->info |= (usb_gettoggle(urb->dev, usb_pipeendpoint(urb->pipe), usb_pipeout(urb->pipe)) <<
TD_TOKEN_TOGGLE);
1031      usb_dotoggle(urb->dev, usb_pipeendpoint(urb->pipe), usb_pipeout(urb->pipe));
1032
1033      urb->status = -EINPROGRESS;
1034 }
```

调用这个函数干什么呢？从代码中可以看出，从这个中断传输的交互队列中找到其（惟一的）交互，将其 TD_CTRL_ACTIVE 标志位（以及 TD_CTRL_IOC 标志位）重新设成 1。此外，也要翻转 tocken 信包中的序号位。前面讲过，USB 控制器在执行中断交互时不将交互描述块脱链，而只是将其 TD_CTRL_ACTIVE 标志位（实际上还有 TD_CTRL_IOC 标志位）清 0，以后执行时就会跳过这个交互。要使 USB 控制器下一次扫描到这个交互描述块时再次执行中断交互，只要把 TD_CTRL_ACTIVE 标志位再设成 1 就行了。所以，如果 interval 非 0，在执行完这个函数后就可以返回了（1431 行），中断传输的周期性就是这样通过硬件与软件的配合而实现的。如果 interval 为 0，则该次中断传输是"一次性"的，所以完成后要释放占用的带宽并从调度队列中脱链。

看到这里，不知读者是否感到有些异样？试想，等时传输和中断传输一样也是周期性的，理应和中断传输受到同样的处理，怎么反倒与控制传输和成块传输为伍呢？这不是站错了队吗？后面我们会回答这个问题。

扫描器和 USB 集中器都不使用等时传输，那是专为实时的音频以及视频设备（如电话、可视电话等等）而设计的。不过，在阅读了其它三种传输的代码以后，再来看等时传输的代码其实是水到渠成的事了。我们只把注意力集中在 uhci_submit_urb()、uhci_submit_isochronous()、uhci_transfer_result()、以及 uhci_result_isochronous()这几个函数上；从驱动程序的角度来说，与其它传输的区别也就在于这几个函数。顺便提一下，与等时传输有关的函数名和常数定义中常常出现"iso"，这表示 isochronous、而跟"国际标准化组织"毫无关系。

我们先看 uhci_submit_urb()中的片断（drivers/usb/uhci.c）：

==================== drivers/usb/uhci.c 1345 1362 ====================
[usb_submit_urb()>uhci_submit_urb()]

```
1345      case PIPE_ISOCHRONOUS:
```

```
1346         if (urb->bandwidth == 0) {  /* not yet checked/allocated */
1347             if (urb->number_of_packets <= 0) {
1348                 ret = -EINVAL;
1349                 break;
1350             }
1351             bustime = usb_check_bandwidth(urb->dev, urb);
1352             if (bustime < 0) {
1353                 ret = bustime;
1354                 break;
1355             }
1356 
1357             ret = uhci_submit_isochronous(urb);
1358             if (ret == -EINPROGRESS)
1359                 usb_claim_bandwidth(urb->dev, urb, bustime, 1);
1360         } else        /* bandwidth is already set */
1361             ret = uhci_submit_isochronous(urb);
1362         break;
```

显然，这与用于中断传输的代码几乎完全是一样的，只不过把 uhci_submit_interrupt()换成了 uhci_submit_isochronous()。这个函数的代码在 drivers/usb/uhci.c：

===================== drivers/usb/uhci.c 1184 1215 =====================

[usb_submit_urb()>uhci_submit_urb()>uhci_submit_isochronous()]

```
1184 static int uhci_submit_isochronous(struct urb *urb)
1185 {
1186     struct uhci_td *td;
1187     struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
1188     int i, ret, framenum;
1189     int status, destination;
1190 
1191     status = TD_CTRL_ACTIVE | TD_CTRL_IOS;
1192     destination = (urb->pipe & PIPE_DEVEP_MASK) | usb_packetid(urb->pipe);
1193 
1194     ret = isochronous_find_start(urb);
1195     if (ret)
1196         return ret;
1197 
1198     framenum = urb->start_frame;
1199     for (i = 0; i < urb->number_of_packets; i++, framenum++) {
1200         if (!urb->iso_frame_desc[i].length)
1201             continue;
1202 
1203         td = uhci_alloc_td(urb->dev);
1204         if (!td)
1205             return -ENOMEM;
1206 
1207         uhci_add_td_to_urb(urb, td);
1208         uhci_fill_td(td, status, destination | ((urb->iso_frame_desc[i].length - 1) << 21),
1209             virt_to_bus(urb->transfer_buffer + urb->iso_frame_desc[i].offset));
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1210
1211         if (i + 1 >= urb->number_of_packets)
1212             td->status |= TD_CTRL_IOC;
1213
1214         uhci_insert_td_frame_list(uhci, td, framenum);
1215     }
```

等时传输中的变互次数完全取决于数据信包的个数。由于只有最后一个信包的 TD_CTRL_IOC 位才设置成 1（见 1211～1212 行），所以要到整个传输结束以后才会引起中断（如果此前没有别的交互引起中断）。

对于 USB 总线，等时信包的内容是无结构的字节流，其各个信包中的数据往往都在同一个大缓冲区中，而只是位移不同，因而不能在各个信包缓冲区之间插入其它信息。所以，要把每个信包缓冲区的起点与长度另外保存在某个地方，为此在 usb 数据结构中设立了一个结构数组 iso_frame_desc[]，用来记录本次传输中各个信包缓冲区的起点与长度。等时传输一般是按“帧”进行的，也叫“ frame”，这里的“ frame_desc”是“帧描述结构”的意思，而并不是指“框架”。具体设备的驱动程序要在调用 usb_submit_urb()之前设置好这个数组。

与中断传输的调度不同，等时传输要落实到具体的（一个或多个）框架上，构成该等时传输请求的各个交互请求要挂入具体框架的队列中去。前面的 usb_check_bandwidth()只是从平均带宽的角度确定了能够满足给定等时传输的要求，但是却并没有落实到具体的框架中。所以，代码中通过 isochronous_find_start()寻找一个框架作为起点，其代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1158 1182 ====================

[usb_submit_urb()>uhci_submit_urb()>uhci_submit_isochronous()>isochronous_find_start()]

```
static int isochronous_find_start(struct urb *urb)
{
    int limits;
    unsigned int start = 0, end = 0;

    if (urb->number_of_packets > 900)    /* 900? Why? */
        return -EFBIG;

    limits = isochronous_find_limits(urb, &start, &end);

    if (urb->transfer_flags & USB_ISO_ASAP) {
        if (limits) {
            int curframe;

            curframe = uhci_get_current_frame_number(urb->dev) % UHCI_NUMFRAMES;
            urb->start_frame = (curframe + 10) % UHCI_NUMFRAMES;
        } else
            urb->start_frame = end;
    } else {
        urb->start_frame %= UHCI_NUMFRAMES;
        /* FIXME: Sanity check */
    }

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    return 0;
}
```

等时传输的信包大小取决于目标设备，但不得超过 1024 字节。当一个等时传输包含多个信包，从而包含多个交互时，要尽量把这些信包分布到不同的框架中，而不是先把一个框架填满（90%）以后，再前进到下一个框架。进一步，对同一目标设备的不同等时传输也应尽量分布到不同的框架中，要避免使一个框架被同一目标设备的交互使用多次。所以，代码中先通过 isochronous_find_limits()看一下对于这个等时传输的起点有否限制。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1123 1156 ====================

[usb_submit_urb()>uhci_submit_urb()>uhci_submit_isochronous()>isochronous_find_start()>isochronous_find_limits()]

```
static int isochronous_find_limits(struct urb *urb, unsigned int *start, unsigned int *end)
{
    struct urb *last_urb = NULL;
    struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
    struct list_head *tmp, *head = &uhci->urb_list;
    int ret = 0;
    unsigned long flags;

    nested_lock(&uhci->urblist_lock, flags);
    tmp = head->next;
    while (tmp != head) {
        struct urb *u = list_entry(tmp, struct urb, urb_list);

        tmp = tmp->next;

        /* look for pending URB's with identical pipe handle */
        if ((urb->pipe == u->pipe) && (urb->dev == u->dev) &&
            (u->status == -EINPROGRESS) && (u != urb)) {
            if (!last_urb)
                *start = u->start_frame;
            last_urb = u;
        }
    }

    if (last_urb) {
        *end = (last_urb->start_frame + last_urb->number_of_packets) & 1023;
        ret = 0;
    } else
        ret = -1;/* no previous urb found */

    nested_unlock(&uhci->urblist_lock, flags);

    return ret;
}
```

我们把它留给读者。这个函数返回 0 时表示对给定等时传输的起点有限制，返回-1 则表示可以任意决定。回到 isochronous_find_start()的代码中，标志位 USB_ISO_ASAP 表示“愈早愈好（as soon as

possible）”；所以。如果可以任意决定就从 USB 主控器的寄存器中读入当前的框架号，在此基础上加 10 作为起点。不过，由于等时交互的执行是周期性的，所谓早晚只对第一次交互有意义。

同到 uhci_submit_isochronous()中，可以看出通过 uhci_insert_td_frame_list()将各个交互请求插入框架时是指定了框架号的，并且框架号逐次递增，其起点就是上面通过 isochronous_find_start()确定的。这样，就使同一传输的各个交互分布到了不同的框架中。

函数 uhci_insert_td_frame_list()的代码在 drivers/usb/uhci.c 中，我们把它留给读者。

==================== drivers/usb/uhci.c 200 220 ====================

[usb_submit_urb()>uhci_submit_urb()>uhci_submit_isochronous()>uhci_insert_td_frame_list()]

```
static void uhci_insert_td_frame_list(struct uhci *uhci, struct uhci_td *td, unsigned framenum)
{
	unsigned long flags;
	struct uhci_td *nexttd;

	framenum %= UHCI_NUMFRAMES;

	spin_lock_irqsave(&uhci->framelist_lock, flags);

	td->frameptr = &uhci->fl->frame[framenum];
	td->link = uhci->fl->frame[framenum];
	if (!(td->link & (UHCI_PTR_TERM | UHCI_PTR_QH))) {
		nexttd = (struct uhci_td *)uhci_ptr_to_virt(td->link);
		td->nexttd = nexttd;
		nexttd->prevtd = td;
		nexttd->frameptr = NULL;
	}
	uhci->fl->frame[framenum] = virt_to_bus(td);

	spin_unlock_irqrestore(&uhci->framelist_lock, flags);
}
```

显然，等时传输的各个交互都是以交互请求为单位插入各个框架，而不像控制交互或成块交互那样是以整个传输为单位将一个队列挂入调度系统中。

USB 控制器对等时交可的执行与中断交互相似。由于等时交互队列物理上并没有队列头，因而等时交互的执行不在“队列上下文”中。这样，主控制器在执行完一个等时交互时便不会推进队列头中的指针，所以不会将等时交互从队列中脱链。

在中断服务程序中调用的 uhci_result_isochronous()与其它传输大同小异。这个函数的代码在 drivers/usb/uhci.c 中，我们把它留给读者。

==================== drivers/usb/uhci.c 1222 1261 ====================

[uhci_interrupt()>uhci_transfer_result()>uhci_result_isochronous()]

```
static int uhci_result_isochronous(struct urb *urb)
{
	struct list_head *tmp, *head;
	struct urb_priv *urbp = (struct urb_priv *)urb->hcpriv;
	int status;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    int i, ret = 0;

    if (!urbp)
        return -EINVAL;

    urb->actual_length = 0;

    i = 0;
    head = &urbp->list;
    tmp = head->next;
    while (tmp != head) {
        struct uhci_td *td = list_entry(tmp, struct uhci_td, list);
        int actlength;

        tmp = tmp->next;

        if (td->status & TD_CTRL_ACTIVE)
            return -EINPROGRESS;

        actlength = uhci_actual_length(td->status);
        urb->iso_frame_desc[i].actual_length = actlength;
        urb->actual_length += actlength;

        status = uhci_map_status(uhci_status_bits(td->status), usb_pipeout(urb->pipe));
        urb->iso_frame_desc[i].status = status;
        if (status != 0) {
            urb->error_count++;
            ret = status;
        }

        i++;
    }

    return ret;
}
```

但是，就如前面所指出的，uhci_transfer_result()中对等时传输的处理有些特殊，或者更确切地说，是应该特殊而没有特殊，把周期性的等时传输同于非周期性的控制传输和成块传输了。我们再来看 uhci_transfer_result()中的有关片段（drivers/usb/uhci.c）：

===================== drivers/usb/uhci.c 1416 1426 =====================

[uhci_interrupt()>uhci_transfer_result()]

```
    switch (usb_pipetype(urb->pipe)) {
    case PIPE_CONTROL:
    case PIPE_BULK:
    case PIPE_ISOCHRONOUS:
        /* Release bandwidth for Interrupt or Isoc. transfers */
        /* Spinlock needed ? */
        if (urb->bandwidth)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1423                usb_release_bandwidth(urb->dev, urb, 1);
1424           uhci_unlink_generic(urb);
1425           break;
1426      case PIPE_INTERRUPT:
    . . . . . .
==================== drivers/usb/uhci.c 1439 1474 ====================
1439           break;
1440      }
1441
1442      if (urb->next) {
1443           turb = urb->next;
1444           do {
1445                if (turb->status != -EINPROGRESS) {
1446                     proceed = 1;
1447                     break;
1448                }
1449
1450                turb = turb->next;
1451           } while (turb && turb != urb && turb != urb->next);
1452
1453           if (turb == urb || turb == urb->next)
1454                is_ring = 1;
1455      }
1456
1457      if (urb->complete && !proceed) {
1458           urb->complete(urb);
1459           if (!proceed && is_ring)
1460                uhci_submit_urb(urb);
1461      }
1462
1463      if (proceed && urb->next) {
1464           turb = urb->next;
1465           do {
1466                if (turb->status != -EINPROGRESS &&
1467                    uhci_submit_urb(turb) != 0)
1468
1469                turb = turb->next;
1470           } while (turb && turb != urb->next);
1471
1472           if (urb->complete)
1473                urb->complete(urb);
1474      }
```

本来，USB 控制器在执行等时交互以后不将其脱链，就是为了保证其周期性。可是，这里仍旧通过 uhci_unlink_generic()将整个等时传输请求从全部有关的队列中脱链，其中也包括从采用物理地址的队列中脱链，这是在 uhci_unlink_generic()中经 uhci_destroy_urb_priv()调用 uhci_remove_td()完成的，读者可以回过去看一下。也就是说，虽然 USB 控制器的硬件不把等时交互从执行队列中脱链，可是软件还是在中断服务程序中把它脱链了。为什么呢？这要与下面 1442～1474 行的代码结合起来看。前面讲

过，这段代码的目的是使等时传输流水线化。我们以电话为例来说明为什么有这个问题。假定有个 IP 电话机接在 USB 总线上，通过等时传输与主机通信，并且主机中的软件和网络接口建立起与对外的连接。为这个目的，在 USB 总线上至少要建立起两个等时传输，分别用于正反两个方向。我们考虑其中从主机到电话机这个方向的等时传输。如果从网上接收到了来自对方的数据，并通过这惟一的一个等时传输发送给电话机，那么电话机每一秒钟才中断一次，每次得到够用一秒钟的数据量。可是，这一来从主机到电话机这一段路上就有了一秒钟的时差。对于有些应用，这固定的一秒钟时差是可以接受的，但是对于电话却是不可接受的。要解决这个问题，就只好把传输的单位分小，例如分成每 50 毫秒一个传输，也就是一秒钟共 20 个传输（尽管是以同一个端点为目标），这样就可以使因为 USB 总线传输而引入的时差降到 200 毫秒。另一方面，即使是可以接受一秒钟时差的应用，也有个从发生中断以后到下一趟执行之间是否来得及处理的问题，所以一般至少也要实行“双缓冲”，即交替使用两个缓冲区的方法。具体到等时传输，就是把整个流量分成两个传输。所以，需要通过等时传输传递的流量总是分成 N 个传输，N 为 1 只是个特例。对于这 N 个传输，usb 结构中提供了一个指针 next，用来把这些传输的 usb 结构连成一个环。每当有一个传输完成而发生中断时，都把这个传输从调度系统中摘下来，供设备驱动程序的高层或应用软件处理，同时就将上一次（或上几次）摘下的传输再提交回调度系统。如此周而复始，就使整个流量分布得均匀了。我们把 1442～1474 行细节留给读者自己阅读。注意凡是已经提交而尚未执行的传输其状态必定是-EINPROGRESS。那么，如果确实只要一个传输就可以了呢？可以让它的指针 next 指出自己，这样在 1460 行就会把它自己再提交给调度系统。

从总体上看，宏观地看，这样的设计当然是很合理（而且很必要）的，但是，从实现的细节看则还可改进。USB 控制器之所以只将在队列上下文中的交互请求从执行队列中摘下，而让不在队列上下文中的交互请求留在队列中，就是因为考虑到了这些交互请求是周期性的。而现在仍要将这些交互请求反复地摘下又挂上，岂不是没有“把政策用足”？再说，只要不将交互清求中的 TD_CTRL_ACTIVE 标志位设成 1，即使留在队列中也没有关系，USB 控制器在执行时如发现一个交互请求处于“不活跃 ”状态就会跳过它继续往前执行。还有，交互请求与缓冲区之间的惟一联系只是一个指针 link，真有必要时也还可以“山不转水转”，让交互请求留在队列中而让缓冲区周转。总之，用于等时传输的这段代码是可以优化的。等时传输的实现还是比较新的，还不像内核的主体部分那样已经经历了无数人的横挑鼻子竖挑眼，还没有来得及走过比较深入的优化、改进的阶段，这也是可以理解的。

不过，尽管有待改进，等时传输的功能还是完整的，并已用于具体的设备。

结合以上的解释，读者应该不难读懂与等时传输有关的这几个函数，从而理解等时传输的机理与过程。可是，不结合具体的设备来看这些代码总令人有点“脱离实际”的感觉，所以我们找了一个摄像机的驱动程序作为实例，使读者可以看到具体的设备怎样建立起 USB 总线上的等时传输，又怎样通过等时传输在应用进程与设备之间传递数据。与这种摄像机有关的代码都在 drivers/usb/ibmcam.c 和 drivers/usb/ibmcam.h 两个文件中，我们在这里引用的只是其中的一部分。

同样，摄像机的驱动程序也是通过系统调用 ioctl()实现对设备的控制，具体的函数为 ibmcam_ioctl()。与前面的扫描器驱动不同的是，对摄像机的驱动基本上是在内核中实现的，所以其“设备驱动层”比我们前面看到的要厚一些，例如在 ibmcam_ioctl()中就实现了不少具体的控制命令。但是，我们在这里并不关心摄像机中的具体资源（例如有些什么寄存器）以及对这些资源的具体操作（例如怎样设置色调等等），而以控制传输为手段来实施这些操作这一个“基本点”则是相同的。所以我们把 ibmcam_ioctl()及有关的代码留给进一步有兴趣的读者。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

这里，我们先看等时传输的建立。这是由 ibmcam_init_isoc()在打开摄像机设备文件时完成的。

==================== drivers/usb/ibmcam.c 2148 2217 ====================

[ibmcam_open(1)>ibmcam_init_isoc()]

```
static int ibmcam_init_isoc(struct usb_ibmcam *ibmcam)
{
	struct usb_device *dev = ibmcam->dev;
	int i, err;

	if (!IBMCAM_IS_OPERATIONAL(ibmcam))
		return -EFAULT;

	ibmcam->compress = 0;
	ibmcam->curframe = -1;
	ibmcam->cursbuf = 0;
	ibmcam->scratchlen = 0;

	/* Alternate interface 1 is is the biggest frame size */
	i = usb_set_interface(dev, ibmcam->iface, ibmcam->ifaceAltActive);
	if (i < 0) {
		printk(KERN_ERR "usb_set_interface error\n");
		ibmcam->last_error = i;
		return -EBUSY;
	}
	usb_ibmcam_change_lighting_conditions(ibmcam);
	usb_ibmcam_set_sharpness(ibmcam);
	usb_ibmcam_reinit_iso(ibmcam, 0);

	/* We double buffer the Iso lists */

	for (i=0; i < IBMCAM_NUMSBUF; i++) {
		int j, k;
		urb_t *urb;

		urb = usb_alloc_urb(FRAMES_PER_DESC);
		if (urb == NULL) {
			printk(KERN_ERR "ibmcam_init_isoc: usb_init_isoc() failed.\n");
			return -ENOMEM;
		}
		ibmcam->sbuf[i].urb = urb;
		urb->dev = dev;
		urb->context = ibmcam;
		urb->pipe = usb_rcvisocpipe(dev, ibmcam->video_endp);
		urb->transfer_flags = USB_ISO_ASAP;
		urb->transfer_buffer = ibmcam->sbuf[i].data;
		urb->complete = ibmcam_isoc_irq;
		urb->number_of_packets = FRAMES_PER_DESC;
		urb->transfer_buffer_length = ibmcam->iso_packet_len * FRAMES_PER_DESC;
		for (j=k=0; j < FRAMES_PER_DESC; j++, k += ibmcam->iso_packet_len) {
```

```
2193             urb->iso_frame_desc[j].offset = k;
2194             urb->iso_frame_desc[j].length = ibmcam->iso_packet_len;
2195         }
2196     }
2197
2198     /* Link URBs into a ring so that they invoke each other infinitely */
2199     for (i=0; i < IBMCAM_NUMSBUF; i++) {
2200         if ((i+1) < IBMCAM_NUMSBUF)
2201             ibmcam->sbuf[i].urb->next = ibmcam->sbuf[i+1].urb;
2202         else
2203             ibmcam->sbuf[i].urb->next = ibmcam->sbuf[0].urb;
2204     }
2205
2206     /* Submit all URBs */
2207     for (i=0; i < IBMCAM_NUMSBUF; i++) {
2208         err = usb_submit_urb(ibmcam->sbuf[i].urb);
2209         if (err)
2210             printk(KERN_ERR "ibmcam_init_isoc: usb_run_isoc(%d) ret %d\n",
2211                 i, err);
2212     }
2213
2214     ibmcam->streaming = 1;
2215     /* printk(KERN_DEBUG "streaming=1 ibmcam->video_endp=$%02x\n", ibmcam->video_endp); */
2216     return 0;
2217 }
```

我们把注意力集中在从 2172～2212 行之间这些代码上。常数 IBMCAM_NUMSBUF 在 drivers/usb/ibmcam.h 中定义为 2，表示采用的是双缓冲，但是显然也可以通过改变其定义而改成多缓冲。对于单向、实时要求又不那么高的视频信号，一般双缓冲就可以了；对交互式的音频信号（如电话）则有更高的要求。代码中通过一个 for 循环为两个缓冲区分别建立起 urb 数据结构，并加以初始化。这里的常数 FRAMES_PER_DESC 定义为 32。从 2186 行可以看出所使用的管道是通过 usb_rcvisocpipe() 取得的，因此是输入方向的等时传输。摄像机没有输出方向的等时传输。此外，从 2189 行可见，摄像机的中断服务程序是 ibmcam_isoc_irq()。

然后，注意 2198～2204 行，这里把这些（两个或更多）urb 数据结构通过结构中的指针 next 连成一个环。

最后，2207～2212 行又通过一个循环提交这些 urb 结构。这样，对于这个摄像机而言，每秒钟将进行两次等时传输，每次包括 32 帧画面，一共是 64 帧，但是也可以通过改变 FRAMES_PER_DESC 的定义加以改变。每当完成其中的一次传输，即 32 帧画面的传递时，就产生一次中断，在上述的 uhci_transfer_result()中先将刚完成的传输请求脱链，再顺着指针 next 找到上一次已经脱链的传输请求（1445 行），并再次提交这个传输请求（1467 行），最后调用摄像机的中断服务程序（1473 行）。

==================== drivers/usb/ibmcam.c 1212 1223 ====================

[uhci_interrupt()>uhci_transfer_result()>ibmcam_isoc_irq()]

```
1212 static void ibmcam_isoc_irq(struct urb *urb)
1213 {
```

```
    int len;
    struct usb_ibmcam *ibmcam = urb->context;
    struct ibmcam_sbuf *sbuf;
    int i;

    /* We don't want to do anything if we are about to be removed! */
    if (!IBMCAM_IS_OPERATIONAL(ibmcam))
        return;

#if 0
 . . . . . .
```

==================== drivers/usb/ibmcam.c 1233 1250 ====================

```
#endif

    if (!ibmcam->streaming) {
        if (debug >= 1)
            printk(KERN_DEBUG "ibmcam: oops, not streaming, but interrupt\n");
        return;
    }

    sbuf = &ibmcam->sbuf[ibmcam->cursbuf];

    /* Copy the data received into our scratch buffer */
    len = ibmcam_compress_isochronous(ibmcam, urb);

    ibmcam->urb_count++;
    ibmcam->urb_length = len;
    ibmcam->data_count += len;

#if 0  /* This code prints few initial bytes of ISO data: used to decode markers */
 . . . . . .
```

==================== drivers/usb/ibmcam.c 1257 1279 ====================

```
#endif

    /* If we collected enough data let's parse! */
    if (ibmcam->scratchlen) {
        /* If we don't have a frame we're current working on, complain */
        if (ibmcam->curframe >= 0)
            ibmcam_parse_data(ibmcam);
        else {
            if (debug >= 1)
                printk(KERN_DEBUG "ibmcam: received data, but no frame available\n");
        }
    }

    for (i = 0; i < FRAMES_PER_DESC; i++) {
        sbuf->urb->iso_frame_desc[i].status = 0;
        sbuf->urb->iso_frame_desc[i].actual_length = 0;
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1274
1275     /* Move to the next sbuf */
1276     ibmcam->cursbuf = (ibmcam->cursbuf + 1) % IBMCAM_NUMSBUF;
1277
1278     return;
1279 }
```

这里主要的操作是 1244 行对 ibmcam_compress_isochronous()的调用，这将把从摄像机读入的原始数据复制到另一个缓冲区中。这些原始数据是经过压缩的，所以还要通过 ibmcam_parse_data()解压缩。解除了压缩的数据就留住缓冲区中，等待应用进程通过系统调用读取。在等时传输中，对缓冲区的写入和读出通常没有互锁和同步，如果应用进程尚未把前一次传输中来自摄像机的数据读走，而后面的数据又来了，就会把前一次的数据覆盖掉。

对摄像机的数据通道只有输入没有输出，所以其系统调用 write()为空操作，而 read()的实现则为 ibmcam_read()。

==================== drivers/usb/ibmcam.c 2743 2821 ====================

```
static long ibmcam_read(struct video_device *dev, char *buf, unsigned long count, int noblock)
{
    struct usb_ibmcam *ibmcam = (struct usb_ibmcam *)dev;
    int frmx = -1;
    volatile struct ibmcam_frame *frame;

    if (debug >= 1)
        printk(KERN_DEBUG "ibmcam_read: %ld bytes, noblock=%d\n", count, noblock);

    if (!IBMCAM_IS_OPERATIONAL(ibmcam) || (buf == NULL))
        return -EFAULT;

    /* See if a frame is completed, then use it. */
    if (ibmcam->frame[0].grabstate >= FRAME_DONE)  /* _DONE or _ERROR */
        frmx = 0;
    else if (ibmcam->frame[1].grabstate >= FRAME_DONE)/* _DONE or _ERROR */
        frmx = 1;

    if (noblock && (frmx == -1))
        return -EAGAIN;

    /* If no FRAME_DONE, look for a FRAME_GRABBING state. */
    /* See if a frame is in process (grabbing), then use it. */
    if (frmx == -1) {
        if (ibmcam->frame[0].grabstate == FRAME_GRABBING)
            frmx = 0;
        else if (ibmcam->frame[1].grabstate == FRAME_GRABBING)
            frmx = 1;
    }

    /* If no frame is active, start one. */
    if (frmx == -1)
```

```
        ibmcam_new_frame(ibmcam, frmx = 0);

    frame = &ibmcam->frame[frmx];

restart:
    if (!IBMCAM_IS_OPERATIONAL(ibmcam))
        return -EIO;
    while (frame->grabstate == FRAME_GRABBING) {
        interruptible_sleep_on((void *)&frame->wq);
        if (signal_pending(current))
            return -EINTR;
    }

    if (frame->grabstate == FRAME_ERROR) {
        frame->bytes_read = 0;
        if (ibmcam_new_frame(ibmcam, frmx))
            printk(KERN_ERR "ibmcam_read: ibmcam_new_frame error\n");
        goto restart;
    }

    if (debug >= 1)
        printk(KERN_DEBUG "ibmcam_read: frmx=%d, bytes_read=%ld, scanlength=%ld\n",
            frmx, frame->bytes_read, frame->scanlength);

    /* copy bytes to user space; we allow for partials reads */
    if ((count + frame->bytes_read) > frame->scanlength)
        count = frame->scanlength - frame->bytes_read;

    if (copy_to_user(buf, frame->data + frame->bytes_read, count))
        return -EFAULT;

    frame->bytes_read += count;
    if (debug >= 1)
        printk(KERN_DEBUG "ibmcam_read: {copy} count used=%ld, new bytes_read=%ld\n",
            count, frame->bytes_read);

    if (frame->bytes_read >= frame->scanlength) { /* All data has been read */
        frame->bytes_read = 0;

        /* Mark it as available to be used again. */
        ibmcam->frame[frmx].grabstate = FRAME_UNUSED;
        if (ibmcam_new_frame(ibmcam, frmx ? 0 : 1))
            printk(KERN_ERR "ibmcam_read: ibmcam_new_frame returned error\n");
    }

    return count;
}
```

这段代码就留给读者了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

最后，我们还要看一下对传输超时的处理。对控制传输和成块传输的调度是不预先分配带宽的，所以有可能虽然把一个传输（特别是成块传输）挂入了队列，却很长时间不能得到执行。另一方面，我们在前面也提到过，如果目标设备一时不能完成所要求的传输，例如要求从扫描器读入数据，可是扫描器老是因为无数据可读而返回 NAK，则也会造成超时。在这些情况下应该有个机制，让等待时间过长的传输请求因超时而失败。为此，在根集中器的初始化过程中通过 rh_init_int_timer()设置了一个定时器，使 CPU 每隔一段时间就来扫描一遍 USB 总线的 urb_list，检查各个传输是否已经超时。这个函数的代码在 drivers/usb/uhci.c 中：

==================== drivers/usb/uhci.c 1774 1787 ====================

```
/* Root Hub INTs are polled by this timer */
static int rh_init_int_timer(struct urb *urb)
{
     struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;

     uhci->rh.interval = urb->interval;
     init_timer(&uhci->rh.rh_int_timer);
     uhci->rh.rh_int_timer.function = rh_int_timer_do;
     uhci->rh.rh_int_timer.data = (unsigned long)urb;
     uhci->rh.rh_int_timer.expires = jiffies + (HZ * (urb->interval < 30 ? 30 : urb->interval)) / 1000;
     add_timer(&uhci->rh.rh_int_timer);

     return 0;
}
```

当定时器到点时，CPU 就会执行 rh_int_timer_do()，其代码就在同一文件中：

==================== drivers/usb/uhci.c 1730 1771 ====================

```
static void rh_int_timer_do(unsigned long ptr)
{
     struct urb *urb = (struct urb *)ptr;
     struct uhci *uhci = (struct uhci *)urb->dev->bus->hcpriv;
     struct list_head *tmp, *head = &uhci->urb_list;
     struct urb_priv *urbp;
     int len;
     unsigned long flags;

     if (uhci->rh.send) {
          len = rh_send_irq(urb);
          if (len > 0) {
               urb->actual_length = len;
               if (urb->complete)
                    urb->complete(urb);
          }
     }

     nested_lock(&uhci->urblist_lock, flags);
     tmp = head->next;
     while (tmp != head) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            struct urb *u = list_entry(tmp, urb_t, urb_list);

            tmp = tmp->next;

            urbp = (struct urb_priv *)u->hcpriv;
            if (urbp) {
                /* Check if the FSBR timed out */
                if (urbp->fsbr && time_after_eq(jiffies, urbp->inserttime + IDLE_TIMEOUT))
                    uhci_fsbr_timeout(uhci, u);

                /* Check if the URB timed out */
                if (u->timeout && time_after_eq(jiffies, u->timeout)) {
                    u->transfer_flags |= USB_ASYNC_UNLINK | USB_TIMEOUT_KILLED;
                    uhci_unlink_urb(u);
                }
            }
        }
        nested_unlock(&uhci->urblist_lock, flags);

        rh_init_int_timer(urb);
}
```

这里通过一个 while 循环扫描 uhci 结构中的 urb_list 队列，检查两种超时。一种是 FSBR 超时。如果一个传输请求要求通过回收每个框架中的剩余时间使其尽快得到执行，而实际上却在 IDLE_TIMEOUT，即 50 毫秒以后仍未得到执行，便说明系统的负荷很大，根本就没有剩余时间可以回收，因此通过 uhci_fsbr_timeout()拆除为此而建立的链接。另一种便是传输本身的超时。读者曾经看到，在提交传输请求时说明了愿意等待的时间。根据当时的时间和愿意等待的时间，就可以计算出在什么时候超时，这就是这里的 u->timeout。如果当前的时间已经过了这一点，那就调用 uhci_unlink_urb()，将这个传输请求从调度系统中脱离出来，并唤醒正在等待的进程，就好像对这个传输请求的执行已经失败了一样。最后，在完成了扫描以后，又调用 rh_init_int_timer()再次设置好定时器，使整个过程周而复始。函数 uhci_unlink_urb()的代码也在 drivers/usb/uhci.c 中，我们就把它留给读者了。

8.10 系统调用 select()以及异步输入／输出

我们以前讲过，一个已打开文件可以是常规文件，也可以是设备文件，还可以是为进程间通信而建立的管道或插口。当已打开文件是常规文件时，从文件的读出宏观上是同步的，只要尚未读到文件的末尾，虽然启动读操作的进程也可能受阻而进入睡眠，等待从磁盘上读入，但是我们知道这个进程在一个有限的短时期以后一定会被唤醒，这个时期的长短在很大程度上是可预测的。如果已经读到了文件的末尾，则更是立即就可知道。可是，对一些外部设备或进程间通信机制的读出就不同了。就拿键盘来说吧，如果一个应用进程因为读键盘（如 getchar()）受阻而进入睡眠，那么我们根本无法预测这个进程什么时候会被唤醒，因为我们无法预测操作人员何时会按键盘。从这个意义上说，对设备的读文件操作有可能完全是异步的。对进程间通信机制的读文件操作也是一样。对于这样的已打开文件，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

我们更倾向于称之为“ I/O 通道”。如果一个进程的工作就是等待来自某一个 I/O 通道的输入并作出反应，那就可以用大家熟悉的无穷 while 循环来实现，那就是：启动读操作，如果没有输入数据就睡眠等待，被唤醒就说明有了输入，读取输入数据并作出反应后再启动读操作，如此循环，直至永远。可是，如果输入有可能来自两个或更多个 I/O 通道呢？我们前面提到过一个用来实现“伪终端”的进程就是一个很好的例子：这个进程一方面监视着键盘（包括鼠标器），一方面监视着一个（或多个）进程间通信管道，无论从哪一个通道有了输入都要马上作出反应。可是，如果应用进程正在睡眠等待键盛输入，而进程间通信管道中却有了数据，则应用进程无法及时读出管道中的数据并作出反应，因为从管道的角度看该进程并不在睡眠等待来自这个管道的输入，从而不会将其喊醒。

在实际应用中。应用软件常常需要同时监视若干个 I/O 通道，等待来自其中任何一个通道的输入数据并作出反应。能够实现这个目标的方法当然是有的，例如让应用进程先通过系统调用 fcntl()把这些通道的 O_NONBLOCK 标志位设成 1，然后再通过系统调用 read()读，此时如果没有数据可读就会立即返回-1，而不会进入睡眠。换言之，就是让应用进程工作于“查询”方式，不断地轮流查询各个通道。这样，从应用进程本身的角度看虽然可以达到目的，但是从系统的角度看却大大降低了系统的效率，在多数情况下是不可接受的。还有个办法是利用进程间通信机制中的报文队列，用一个报文队列来作为输入数据的汇集点。就好像信箱一样：信箱只有一个，信件却可以来自四面八方，全部汇集到一个信箱中（所以在有的系统中把报文队列称作“信箱”）。但是，Linux 内核中的设备驱动程序并不能直接将输入数据转换成一个报文，并投递到报文队列中，所以还得为键盘输入设立一个服务进程。这个进程就监视键盘（包括鼠标器）一个通道，只要有输入就把它转换成报文，并发送到指定的报文队列中。在相反的方向上，对来自进程间通信管道的数据也可以作相似的处理。这样固然也能达到要求，但是系统中多了两个进程，增加了进程调度的负担，也会降低系统的效率（不过不像前一个方法那么严重）。

因此，比较理想的办法是让进程的单一目标的睡眠等待变成多目标的睡眠等待。如果能这样，则在我们这个例子中只要应用进程说明等待的目标是那几个通道就行了。当这几个通道中的任何一个有输入数据时，相应的设备驱动程序就会把睡眠中的进程唤醒。这样，既达到了上述的目标，又保持了较高的效率。

同样的问题也存在于输出操作，因为有时候需要为输出而同时等待若干个通道，等待这些通道中的某一个或几个满足可以进行输出操作的条件。

Linux/Unix 为此提供了一个系统调用 select()，内核中的实现为 sys_select()，其代码在 fs/select.c 中：

==================== fs/select.c 257 346 ====================

```
asmlinkage long
sys_select(int n, fd_set *inp, fd_set *outp, fd_set *exp, struct timeval *tvp)
{
	fd_set_bits fds;
	char *bits;
	long timeout;
	int ret, size;

	timeout = MAX_SCHEDULE_TIMEOUT;
	if (tvp) {
		time_t sec, usec;

```

```
        if ((ret = verify_area(VERIFY_READ, tvp, sizeof(*tvp)))
            || (ret = __get_user(sec, &tvp->tv_sec))
            || (ret = __get_user(usec, &tvp->tv_usec)))
             goto out_nofds;

        ret = -EINVAL;
        if (sec < 0 || usec < 0)
             goto out_nofds;

        if ((unsigned long) sec < MAX_SELECT_SECONDS) {
             timeout = ROUND_UP(usec, 1000000/HZ);
             timeout += sec * (unsigned long) HZ;
        }
    }

    ret = -EINVAL;
    if (n < 0)
        goto out_nofds;

    if (n > current->files->max_fdset)
        n = current->files->max_fdset;

    /*
     * We need 6 bitmaps (in/out/ex for both incoming and outgoing),
     * since we used fdset we need to allocate memory in units of
     * long-words.
     */
    ret = -ENOMEM;
    size = FDS_BYTES(n);
    bits = select_bits_alloc(size);
    if (!bits)
        goto out_nofds;
    fds.in     = (unsigned long *)  bits;
    fds.out    = (unsigned long *) (bits +   size);
    fds.ex     = (unsigned long *) (bits + 2*size);
    fds.res_in  = (unsigned long *) (bits + 3*size);
    fds.res_out = (unsigned long *) (bits + 4*size);
    fds.res_ex  = (unsigned long *) (bits + 5*size);

    if ((ret = get_fd_set(n, inp, fds.in)) ||
       (ret = get_fd_set(n, outp, fds.out)) ||
       (ret = get_fd_set(n, exp, fds.ex)))
        goto out;
    zero_fd_set(n, fds.res_in);
    zero_fd_set(n, fds.res_out);
    zero_fd_set(n, fds.res_ex);

    ret = do_select(n, &fds, &timeout);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
318     if (tvp && !(current->personality & STICKY_TIMEOUTS)) {
319         time_t sec = 0, usec = 0;
320         if (timeout) {
321             sec = timeout / HZ;
322             usec = timeout % HZ;
323             usec *= (1000000/HZ);
324         }
325         put_user(sec, &tvp->tv_sec);
326         put_user(usec, &tvp->tv_usec);
327     }
328
329     if (ret < 0)
330         goto out;
331     if (!ret) {
332         ret = -ERESTARTNOHAND;
333         if (signal_pending(current))
334             goto out;
335         ret = 0;
336     }
337
338     set_fd_set(n, inp, fds.res_in);
339     set_fd_set(n, outp, fds.res_out);
340     set_fd_set(n, exp, fds.res_ex);
341
342 out:
343     select_bits_free(bits, size);
344 out_nofds:
345     return ret;
346 }
```

参数 tvp 指向一个 timeval 数据结构，表明准备睡眠等待的最长时间，如果指针为 0 就表示无限期的睡眠等待；从系统调用返回时，这个数据结构表明还剩下未用完的睡眠时间。参数 inp、outp、exp 都是指向 fd_set 数据结构的指针，这种数据结构是关于已打开文件的位图，位图中的每一位都代表着当前进程的一个已打开文件。早期的系统中都用一个 32 位长字作为已打开文件位图，因为那时候一个进程最多只能有 32 个已打开文件，后来则改成可以灵活定义的 fd_set 数据结构。

调用 sys_select()时参数 inp 所指的位图表示当前进程在睡眠中要等待来自哪一些已打开文件的输入；返回时则表明哪些已打开文件中已经有了输入。类似地，outp 表示当前进程在睡眠中要等待对哪一些已打开文件的写操作；返回时则表明对哪一些已打开文件的写操作已可立即进行。至于 exp，则用来临视在哪一些通道中发生了异常。最后，参数 n 表示调用时的参数表中有几个位图。所有这些位图以及 timeval 数据结构都在用户空间，所以先要通过__get_user()和__copy_from_user()从用户空间把这些数据结构复制到内核中，返回时则要反过来通过 put_user()和__copy_to_user()从内核中复制到用户空间。读者对这些代码应该已经熟悉了。

由于 3 个位图各自都有“要求”和“结果”两个版本，在操作的过程中一共需要 6 个位图。所以，先要分配一小块空间用于这 6 个位图，并通过 get_fd_set()把 3 个“要求”位图从用户空间复制过来(308～310 行）。函数 get_fd_set()的代码在 include/linux/poll.h 中：

==================== include/linux/poll.h 52 71 ====================

[sys_select()>set_fd_set()]

```
/*
 * We do a VERIFY_WRITE here even though we are only reading this time:
 * we'll write to it eventually..
 *
 * Use "unsigned long" accesses to let user-mode fd_set's be long-aligned.
 */
static inline
int get_fd_set(unsigned long nr, void *ufdset, unsigned long *fdset)
{
     nr = FDS_BYTES(nr);
     if (ufdset) {
          int error;
          error = verify_area(VERIFY_WRITE, ufdset, nr);
          if (!error && __copy_from_user(fdset, ufdset, nr))
               error = -EFAULT;
          return error;
     }
     memset(fdset, 0, nr);
     return 0;
}
```

操作的主体是 do_select()，其代码在 fs/select.c 中，我们分段阅读。

==================== fs/select.c 163 181 ====================

[sys_select()>do_select()]

```
163 int do_select(int n, fd_set_bits *fds, long *timeout)
164 {
165      poll_table table, *wait;
166      int retval, i, off;
167      long __timeout = *timeout;
168 
169      read_lock(&current->files->file_lock);
170      retval = max_select_fd(n, fds);
171      read_unlock(&current->files->file_lock);
172 
173      if (retval < 0)
174           return retval;
175      n = retval;
176 
177      poll_initwait(&table);
178      wait = &table;
179      if (!__timeout)
180           wait = NULL;
181      retval = 0;
```

参数 fds 是个指针，指向一个 fd_set_bits 结构，结构中就是 6 个工作位图，其中前 3 个为“要求 ”位图。170 行通过 max_select_fd()根据这 3 个位图计算出本次操作所涉及最大的已打开文件号是什么，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

所有号码高于这个数值的已打开文件都与本次操作无关。

这里用到的数据结构 poll_table 定义于 include/linux/poll.h：

==================== include/linux/poll.h 15 18 ====================

```
typedef struct poll_table_struct {
	int error;
	struct poll_table_page * table;
} poll_table;
```

其主体 poll_table_page 定义于 fs/select.c：

==================== fs/select.c 27 37 ====================

```
struct poll_table_entry {
	struct file * filp;
	wait_queue_t wait;
	wait_queue_head_t * wait_address;
};

struct poll_table_page {
	struct poll_table_page * next;
	struct poll_table_entry * entry;
	struct poll_table_entry entries[0];
};
```

当一个进程要进入睡眠，而想要某个设备的驱动程序在设备的状态发生变化时将其唤醒，就得要准备好一个 wait_queue_t 数据结构，并将这个数据结构挂入目标设备的某个等待队列中。读者以前曾多次看到，在等待对象单一时一般都把 wait_queue_t 数据结构建立在堆栈中。可是，在有多个等待对象时就不能那样了。另一方面，在有多个等待对象、从而有多个 wait_queue_t 数据结构时，要有个既有效、又灵活，便于扩充的方法将这些 wait_queue_t 结构管理起来，上面这些数据结构就正是为此而设计的。这里的 poll_table_entry 数据结构既是对 wait_queue_t 的扩充，又是对它的“包装”。此外，poll_table_page 结构中数组 entries[]的下标为 0，表示该数组的大小可以动态地确定。实际使用时总是分配一个页面，页面中能容纳几个 poll_table_entry 结构，这个数组就是多大。使用中指针 entry 总是指向 entries[]中的第一个空闲的 poll_table_entry 结构，根据需要动态地分配 entries[]中的表项。一个页面用完了，就再分配一个，通过指针 next 连成一条单链。

函数 do_select()中定义了一个局部的 poll_table 数据结构 table，177 行先对其进行初始化（include/linux/poll.h）：

==================== include/linux/poll.h 28 32 ====================

[sys_select()>do_select()>poll_initwait()]

```
static inline void poll_initwait(poll_table* pt)
{
	pt->error = 0;
	pt->table = NULL;
}
```

现在的 poll_table 还只是个空架子，还没有为其分配任何页面，因为还不知道到底需要多少。完成

PDF-XCHANGE Click to buy NOW! www.docu-track.com

了这些准备以后，CPU 就进入了一个无穷 for 循环，正常情况下一直要到监视中的某个已打开文件中有了输入或满足了其它等待条件，或者指定的睡眠等待时间已经到期，或者当前进程接收到了信号时才会结束。

==================== fs/select.c 182 234 ====================

[sys_select()>do_select()]

```
	for (;;) {
		set_current_state(TASK_INTERRUPTIBLE);
		for (i = 0 ; i < n; i++) {
			unsigned long bit = BIT(i);
			unsigned long mask;
			struct file *file;

			off = i / __NFDBITS;
			if (!(bit & BITS(fds, off)))
				continue;
			file = fget(i);
			mask = POLLNVAL;
			if (file) {
				mask = DEFAULT_POLLMASK;
				if (file->f_op && file->f_op->poll)
					mask = file->f_op->poll(file, wait);
				fput(file);
			}
			if ((mask & POLLIN_SET) && ISSET(bit, __IN(fds,off))) {
				SET(bit, __RES_IN(fds,off));
				retval++;
				wait = NULL;
			}
			if ((mask & POLLOUT_SET) && ISSET(bit, __OUT(fds,off))) {
				SET(bit, __RES_OUT(fds,off));
				retval++;
				wait = NULL;
			}
			if ((mask & POLLEX_SET) && ISSET(bit, __EX(fds,off))) {
				SET(bit, __RES_EX(fds,off));
				retval++;
				wait = NULL;
			}
		}
		wait = NULL;
		if (retval || !__timeout || signal_pending(current))
			break;
		if(table.error) {
			retval = table.error;
			break;
		}
		__timeout = schedule_timeout(__timeout);
```

```
224         }
225         current->state = TASK_RUNNING;
226 
227         poll_freewait(&table);
228 
229         /*
230          * Up-to-date the caller timeout.
231          */
232         *timeout = __timeout;
233         return retval;
234     }
```

在这个无穷 for 循环（182～224 行）中，CPU 通过另一个内层 for 循环（184～215 行）对“监视位图”进行一次扫描，这里的宏操作 BITS()定义于 fs/select.c：

==================== fs/select.c 112 112 ====================

```
112  #define BITS(fds, n)        (*__IN(fds, n)|*__OUT(fds, n)|*__EX(fds, n))
```

如果三个位图之一中的某一位为 1，就对相应的已打开文件作一次询问（197 行），并把询问的结果汇集到 fds 所指的 fd_set_bits 数据结构中（200～214 行）。一趟扫描下来以后，就检查一下上述的条件是否已经满足，或出了错。如果没有就通过 schedule_timeout()进入睡眠，到被唤醒时再在下一轮循环中作另一次扫描。就这样，除第一次以外，以后都是在进程被唤醒时才执行一遍循环，所以从本质上讲是一种 do-while 循环。

那么，在什么情况下会被唤醒呢？首先，受到询问的已打开文件的设备驱动程序会把当前进程通过一个 wait_queue_t 数据结构，从而 poll_table_entry 数据结构，挂入其唤醒队列，使得该设备的中断服务程序在接收到输入时就会唤醒这个进程。其次，如果指定了时间限制，则当时间到点时也会唤醒这个进程，这是因为进程在进入睡眠时都指定了需要继续睡眠的时间。最后，如果进程接收到了信号也会被唤醒。

显然，这里的关键在于对具体已打开文件，即设备的询问。从代码中可以看出，这是通过具体 file_operations 数据结构中提供的函数指针 poll 进行的。我们以前都把注意力集中在 open、read、write 等更为常用的操作上，有意忽略了这个函数指针，现在要回过来关注这个操作了。另一方面，阅读 poll 操作的代码在某种意义上也是对有关内容的一次复习。

先看对进程间通信管道的 poll 操作，管道文件读端的 file_operations 数据结构定义于 fs/pipe.c 中，我们再回顾一下：

==================== fs/pipe.c 412 412 ====================

```
412  struct file_operations read_pipe_fops = {
     . . . . . .
```

==================== fs/pipe.c 416 416 ====================

```
416         poll:          pipe_poll,
     . . . . . .
```

==================== fs/pipe.c 420 420 ====================

```
420  };
```

可见，管道文件读端的 poll 操作是由 pipe_poll()完成的，其代码也在 fs/pipe.c 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/pipe.c 278 297 ====================

[sys_select()>do_select()>pipe_poll()]

```
/* No kernel lock held - fine */
static unsigned int
pipe_poll(struct file *filp, poll_table *wait)
{
	unsigned int mask;
	struct inode *inode = filp->f_dentry->d_inode;

	poll_wait(filp, PIPE_WAIT(*inode), wait);

	/* Reading only -- no need for acquiring the semaphore.  */
	mask = POLLIN | POLLRDNORM;
	if (PIPE_EMPTY(*inode))
		mask = POLLOUT | POLLWRNORM;
	if (!PIPE_WRITERS(*inode) && filp->f_version != PIPE_WCOUNTER(*inode))
		mask |= POLLHUP;
	if (!PIPE_READERS(*inode))
		mask |= POLLERR;

	return mask;
}
```

管道文件的 inode 结构中有个等待队列的队头，凡是等待着对这个管道进行操作的进程都通过一个 wait_queue_t 数据结构挂入这个队列。当管道的状态发生变化时，若有数据到来或原来满的缓冲区变空时，就会唤醒挂在这个队列中的进程。

宏操作 PIPE_WAIT()返回这个队头的地址，定义于 include/linux/pipe_fs_i.h：

==================== include/linux/pipe_fs_i.h 22 22 ====================

```
#define PIPE_WAIT(inode)	(&(inode).i_pipe->wait)
```

这里的 poll_wait()是个 inline 函数，定义于 include/linux/poll.h：

==================== include/linux/poll.h 22 26 ====================

[sys_select()>do_select()>pipe_poll()>poll_wait()]

```
extern inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
	if (p && wait_address)
		__pollwait(filp, wait_address, p);
}
```

操作的主体是__pollwait()，这个函数的代码在 fs/select.c 中：

==================== fs/select.c 74 103 ====================

[sys_setect()>do_select()>pipe_pull()>poll_wait()>__pollwait()]

```
void __pollwait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
{
	struct poll_table_page *table = p->table;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
78       if (!table || POLL_TABLE_FULL(table)) {
79            struct poll_table_page *new_table;
80
81            new_table = (struct poll_table_page *) __get_free_page(GFP_KERNEL);
82            if (!new_table) {
83                 p->error = -ENOMEM;
84                 __set_current_state(TASK_RUNNING);
85                 return;
86            }
87            new_table->entry = new_table->entries;
88            new_table->next = table;
89            p->table = new_table;
90            table = new_table;
91       }
92
93       /* Add a new entry */
94       {
95            struct poll_table_entry * entry = table->entry;
96            table->entry = entry+1;
97            get_file(filp);
98            entry->filp = filp;
99            entry->wait_address = wait_address;
100           init_waitqueue_entry(&entry->wait, current);
101           add_wait_queue(wait_address,&entry->wait);
102      }
103  }
```

如果还没有用于 poll_table_page 结构的页面，或者最后分配的页面已经用完而不再有空闲的 poll_table_entry 结构，就要为其分配一个新的页面，扩充其容量。

然后将 poll_table_page 结构中的下一个 poll_table_entry 数据结构用作连接件，通过它内部的 wait_queue_t 结构 wait 挂入等待队列 wait_address，而这个等待队列的队头正是在目标文件（通道）的 inode 结构中。在 wait_queue_t 结构中包含着指向当前进程 task_struct 结构的指针，这是在 100 行通过 init_waitqueue_entry()设置的，所以当要将队列中的进程唤醒时从 wait_queue_t 结构开始一下子就能找到相应进程的 task_struct 结构。

以前我们常常看到：当一个进程受阻的时候，一旦把 wait_queue_t 结构挂入一个等待队列，马上就会通过 schedule()进入睡眠。内核代码中还有一组宏操作 wait_event()和 wait_event_interruptible()，把这二者组合在了一起（见第 4 章），以致人们往往不知不觉地把挂入等待队列与进入睡眠划上了等号。其实，一个进程通过一个 wait_queue_t 结构挂入一个等待队列只是向队列的“主人”挂上一个号，让它在发生某种事件时将其唤醒，而到底是否真的入睡则仍有自由。“唤醒”一个本来就醒着的进程并无什么损害。在 do_select()操作中，就把这二者区分开来了。这里，每当查询一个通道时，如果这个通道没有数据可读，就都要向该通道挂上一个号，通过 add_wait_queue()把一个 wait_queue_t 结构挂入该通道的等待对列，但是并不马上进入睡眠；而是要到对所有的通道都查询了一遍以后，回到 do_select()中，如果确实需要睡眠才会真的进入睡眠。

当然，如果最后并没有睡眠，或者睡眠以后被其中的一个（或几个）通道唤醒，总不能让这些

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

wait_queue_t 结构留在各个等待队列中，所以 do_select()中的 227 行要通过 poll_freewait()把所有这些数据结构从各个队列中摘下来（见后）。

下面我们就来看进程的唤醒，先看当管道的写端进程往管道中写入，从而使得读端有了输入时的情景。读者在第 6 章中读过 pipe_write()的代码，我们在这里再把其中与唤醒进程有关的片段摘录于下（fs/pipe.c）：

==================== fs/pipe.c 134 136 ====================

```
134  static ssize_t
135  pipe_write(struct file *filp, const char *buf, size_t count, loff_t *ppos)
136  {
     . . . . . .
```

==================== fs/pipe.c 209 228 ====================

```
209           do {
210                /*
211                 * Synchronous wake-up: it knows that this process
212                 * is going to give up this CPU, so it doesnt have
213                 * to do idle reschedules.
214                 */
215                wake_up_interruptible_sync(PIPE_WAIT(*inode));
216                PIPE_WAITING_WRITERS(*inode)++;
217                pipe_wait(inode);
218                PIPE_WAITING_WRITERS(*inode)--;
219                if (signal_pending(current))
220                     goto out;
221                if (!PIPE_READERS(*inode))
222                     goto sigpipe;
223           } while (!PIPE_FREE(*inode));
224           ret = -EFAULT;
225      }
226
227      /* Signal readers asynchronously that there is more data.  */
228      wake_up_interruptible(PIPE_WAIT(*inode));
     . . . . . .
```

==================== fs/pipe.c 246 246 ====================

```
246  }
```

至于具体的唤醒过程，读者已经在第 4 章中阅读过有关的代码，我们就不重复了。这里只是指出，在 pipe_write()中唤醒的正是在队列 PIPE_WAIT(*inode)中的进程，也就是这个管道文件的 inode 数据结构中的等待队列，与前面读端进程所在的队列是一致的。另一方面，唤醒这个队列中的进程时，从队列中的 wait_queue_t 结构就直接可以找到相应进程的 task_struct 数据结构，而与是否有 poll_table_entry 和 poll_table_page 等数据结构的存在并无关系。

再来看另一个实例，我们假定监视中的另一个通道是鼠标器。同样，也是从这个设备的 file_operations 数据结构 psaux_fops 开始，这个数据结构定义于 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 994 1001 ====================

```
994  struct file_operations psaux_fops = {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        read:         read_aux,
        write:        write_aux,
        poll:         aux_poll,
        open:         open_aux,
        release:  release_aux,
        fasync:       fasync_aux,
};
```

这个设备的查询操作是由 aux_poll()完成的，其代码在同一文件（drivers/char/pc_keyb.c）中：

==================== drivers/char/pc_keyb.c 985 992 ====================

[sys_select()>do_select()>aux_poll()]

```
/* No kernel lock held - fine */
static unsigned int aux_poll(struct file *file, poll_table * wait)
{
    poll_wait(file, &queue->proc_list, wait);
    if (!queue_empty())
        return POLLIN | POLLRDNORM;
    return 0;
}
```

可见，这里同样也调用 poll_wait()，不同的是这一次等待队列不在 inode 数据结构中，而在更底层的设备驱动程序中，具体地说是 queue->proc_list。这里的指针 queue 是 drivers/char/pc_keyb.c 的静态变量，指向一个 aux_queue 数据结构，定义于 drivers/char/pc_keyb.c：

==================== include/linux/pc_keyb.h 124 130 ====================

```
            timeout--;
        } while (timeout);
#ifdef KBD_REPORT_TIMEOUTS
        printk(KERN_WARNING "Keyboard timed out[1]\n");
#endif
}

```

结构中有个成分就是 proc_list，这是个等待队列头，即 wait_queue_head_t 数据结构。所以，把等待队列放在什么地方是由具体的驱动程序自己决定的，只要它自己知道在哪里等待就到哪里去唤醒就可以了。前一个例子中之所以放在 inode 结构中是因为管道没有更底层的设备驱动程序了。那么，鼠标器的驱动程序是否真的到 queue->proc_list 中去唤醒进程呢？当然，下面是 drivers/char/pc_keyb.c 中 handle_mouse_event()的片段：

==================== drivers/char/pc_keyb.c 396 398 ====================

[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()]

```
static inline void handle_mouse_event(unsigned char scancode)
{
#ifdef CONFIG_PSMOUSE
     . . . . . .
```

==================== drivers/char/pc_keyb.c 418 422 ====================

```
            if (head != queue->tail) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                queue->head = head;
                kill_fasync(&queue->fasync, SIGIO, POLL_IN);
                wake_up_interruptible(&queue->proc_list);
            }
```

==================== drivers/char/pc_keyb.c 424 425 ====================

```
#endif
}
```

从某种意义上说，do_select()的策略有点像是“广种薄收”，四处登记要人们在发生什么事时将其唤醒，但是实际上只要有一个通道真的发生了什么而将其唤醒也就可以了。不过，唤醒以后还得对所有的通道再轮询一遍，因为可能有不止一个通道同时唤醒了这个进程。通过轮询，根据所得到的信息设置 3 个“结果”位图并加以计数，就可以知道到底有几个通道唤醒了它以及各自是为什么原因。这就是 do_select()中的外层 for 循环的作用。最后，在正常情况下通过 218 行的 break 语句跳出这个无穷 for 循环。

不管有几个通道唤醒了睡眠中的进程，既然唤醒了，不需要再继续睡眠了，就要把所有的 wait_queue_t 结构从各个等待队列中摘下来，这是由 poll_freewait()完成的。其代码在 fs/select.c 中：

==================== fs/select.c 55 72 ====================

[sys_select()>do_select()>poll_freewait()]

```
void poll_freewait(poll_table* pt)
{
    struct poll_table_page * p = pt->table;
    while (p) {
        struct poll_table_entry * entry;
        struct poll_table_page *old;

        entry = p->entry;
        do {
            entry--;
            remove_wait_queue(entry->wait_address,&entry->wait);
            fput(entry->filp);
        } while (entry > p->entries);
        old = p;
        p = p->next;
        free_page((unsigned long) old);
    }
}
```

我们把这段代码留给读者。通过这段代码，读者应该能体会到为什么要通过 poll_table_entry、poll_table_page、poll_table 等数据结构把所有的 wait_queue_t 结构组装在一起的理由。

回到 sys_select()的代码中，还要通过 set_fd_set()把 3 个“结果”位图复制到用户空间（338～340 行）。函数 set_fd_set()的代码在 include/linux/poll.h 中：

==================== include/linux/poll.h 73 78 ====================

[sys_select()>set_fd_set()]

```
static inline
void set_fd_set(unsigned long nr, void *ufdset, unsigned long *fdset)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
	if (ufdset)
		__copy_to_user(ufdset, fdset, FDS_BYTES(nr));
}
```

最后，通过 select_bits_free()释放当初分配用于 6 个工作位图的空间。

我们以前讲过，许多外部设备的读操作本质上是异步的，因为我们事先无法估计这些设备在什么时候有数据可读。另一方面，到现在为止，我们看到的读操作从进程调度的角度看都是同步的，因为当操作受阻的时候进程就进入睡眠等待，这样才能不浪费 CPU 的资源。一般的设备驱动程序就是这两种不同性质操作的交汇点，或者转换点。设备驱动程序的底层是异步的中断机制，上层却是同步的睡眠 / 唤醒机制。系统调用 select()把这种转换从一个通道推广到了多个通道，但是并没有改变读操作在顶层的同步性。这样，从整个系统的角度看是避免了效率的降低，但是从进程本身的角度看却未必尽善尽美，因为有时候一个进程也许既想监视若干个通道，同时又想进行一些“后台”计算，只是在监视中的某个通道有事件发生（例如有了输入数据）时才转到“前台”作出反应。那么，有没有办法使顶层，即进程的读文件操作也变成异步呢？其实，进程本来就有类似于中断的异步操作机制，那就是“信号”。所以，只要把上层的信号机制与底层的中断机制结合起来，就能在上层实现对设备的异步操作，Linux 内核提供了这样的功能。

需要对设备的异步操作时，一个进程必须作好下列准备：

(1) 先打开目标设备。

(2) 设置好对目标设备的 SIGIO 信号处理程序，并设置好信号响应（见第 6 章）。

(3) 通过系统调用 fcntl()将本进程设置成目标通道（已打开文件）的“主人”。

(4) 通过系统调用 ioctl()将目标通道（已打开文件）设置成异步操作模式。

这样，当目标设备的通道中状态发生变化时，就会向进程发出一个 SIGIO 信号。平时这个进程可以从事“后台”计算，当接收到 SIGIO 信号时就转入前台的信号处理程序中，完成对设备的具体操作。显然，此时的进程就好像受到外部设备中断的 CPU 一样。

下面我们来看这是怎样实现的。首先，file 结构内部有个成分 f_owner，是一个 fown_struct 数据结构，定义于 include/linux/fs.h：

==================== include/linux/fs.h 492 496 ====================

```
struct fown_struct {
	int pid;		/* pid or -pgrp where SIGIO should be sent */
	uid_t uid, euid;	/* uid/euid of process setting the owner */
	int signum;		/* posix.1b rt signal to be delivered on IO */
};
```

这个数据结构的内容可以通过系统调用 fcntl()设置，下面是 sys_fcntl()的主体 do_fcntl()中的片段（fs/fcntl.c）：

==================== fs/fcntl.c 229 234 ====================

[sys_fcntl()>do_fcntl()]

```
static long do_fcntl(unsigned int fd, unsigned int cmd,
		     unsigned long arg, struct file * filp)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
{
        long err = -EINVAL;

        switch (cmd) {
        . . . . . .
```

==================== fs/fcntl.c 273 282 ====================

```
                case F_SETOWN:
                        lock_kernel();
                        filp->f_owner.pid = arg;
                        filp->f_owner.uid = current->uid;
                        filp->f_owner.euid = current->euid;
                        err = 0;
                        if (S_ISSOCK (filp->f_dentry->d_inode->i_mode))
                                err = sock_fcntl (filp, F_SETOWN, arg);
                        unlock_kernel();
                        break;
        . . . . . .
```

==================== fs/fcntl.c 309 312 ====================

```
        }

        return err;
}
```

有些设备驱动程序在通过系统调用 ioctl()设置异步操作模式时会自动设置 f_owner，所以这一步往往并非必须。除此以外，对发送的信号 signum 也可以通过系统调用 fcntl()另加指定，如果不加指定就是 SIGIO。

再看 sys_ioctl()的片段，取自文件 fs/ioctl.c：

==================== fs/ioctl.c 48 54 ====================

```
asmlinkage long sys_ioctl(unsigned int fd, unsigned int cmd, unsigned long arg)
{
        struct file * filp;
        unsigned int flag;
        int on, error = -EBADF;

        filp = fget(fd);
        . . . . . .
```

==================== fs/ioctl.c 58 59 ====================

```
        lock_kernel();
        switch (cmd) {
        . . . . . .
```

==================== fs/ioctl.c 83 101 ====================

```
                case FIOASYNC:
                        if ((error = get_user(on, (int *)arg)) != 0)
                                break;
                        flag = on ? FASYNC : 0;

                        /* Did FASYNC state change ? */
```

```
                if ((flag ^ filp->f_flags) & FASYNC) {
                    if (filp->f_op && filp->f_op->fasync)
                        error = filp->f_op->fasync(fd, filp, on);
                    else error = -ENOTTY;
                }
                if (error != 0)
                    break;

                if (on)
                    filp->f_flags |= FASYNC;
                else
                    filp->f_flags &= ~FASYNC;
                break;
    . . . . . .
```

==================== fs/ioctl.c 109 115 ====================a

```
        }
        unlock_kernel();
        fput(filp);

out:
        return error;
}
```

进行 ioctl()系统调用时，参数 cmd 表示具体的“命令”代码，而 arg 则为对具体命令的参数。对于异步模式的设置，命令码 cmd 为 FIOASYNC，而参数 arg 可以是 1 或 0。

并非所有的设备都支持异步模式。支持异步模式的一般都是与人机界面有关的字符设备，这些设备的 file_operations 结构中的函数指针 fasync 指向其用来建立异步模式的函数。以前面所列鼠标器的 file_operations 结构 psaux_fops 为例，其指针 fasync 指向 fasync_aux()，它的代码在 drivers/char/pc_keyb.c 中：

==================== drivers/char/pc_keyb.c 860 868 ====================

[sys_ioctl()>fasync_aux()]

```
static int fasync_aux(int fd, struct file *filp, int on)
{
        int retval;

        retval = fasync_helper(fd, filp, on, &queue->fasync);
        if (retval < 0)
            return retval;
        return 0;
}
```

这里的全局量 queue 就指向前面看到过的 aux_queue 数据结构，里面有个等待队列 proc_list 用于进程的睡眠 / 唤醒机制，这我们已经看到过了。但是，这个数据结构中同时还有个 fasync_struct 数据结构指针 fasync，用来维持一个单链的“异步文件”队列（fasync 队列）。函数 fasync_helper()的作用就是为当前进程创建一个 fasync_struct 数据结构，并将其挂入目标设备的 fasync 队列。这样，当目标设备的通道中发生某些状态变化时，就可以顺着这个队列给每个有关的进程都发一个 SIGIO 信号。这个函

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

数的代码在 fs/fcntl.c 中：

==================== fs/fcntl.c 438 480 ====================
[sys_ioctl()>fasync_aux()>fasync_helper()]

```
/*
 * fasync_helper() is used by some character device drivers (mainly mice)
 * to set up the fasync queue. It returns negative on error, 0 if it did
 * no changes and positive if it added/deleted the entry.
 */
int fasync_helper(int fd, struct file * filp, int on, struct fasync_struct **fapp)
{
	struct fasync_struct *fa, **fp;
	struct fasync_struct *new = NULL;
	int result = 0;

	if (on) {
		new = kmem_cache_alloc(fasync_cache, SLAB_KERNEL);
		if (!new)
			return -ENOMEM;
	}
	write_lock_irq(&fasync_lock);
	for (fp = fapp; (fa = *fp) != NULL; fp = &fa->fa_next) {
		if (fa->fa_file == filp) {
			if(on) {
				fa->fa_fd = fd;
				kmem_cache_free(fasync_cache, new);
			} else {
				*fp = fa->fa_next;
				kmem_cache_free(fasync_cache, fa);
				result = 1;
			}
			goto out;
		}
	}

	if (on) {
		new->magic = FASYNC_MAGIC;
		new->fa_file = filp;
		new->fa_fd = fd;
		new->fa_next = *fapp;
		*fapp = new;
		result = 1;
	}
out:
	write_unlock_irq(&fasync_lock);
	return result;
}
```

数据结构（类型）fasync_struct 的定义在 include/linux/fs.h 中：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/fs.h 597 602 ====================

```
struct fasync_struct {
	int	magic;
	int	fa_fd;
	struct	fasync_struct	*fa_next; /* singly linked list */
	struct	file		*fa_file;
};
```

这个数据结构中并没有指向 task_struct 结构的指针，但是有指向 file 结构的指针；file 结构代表着进程与文件的连接，找到了 file 结构就可以找到它的“主人”。至于 fasync_helper()的代码，既然这么简单，就不用多说了。

前面，在 handle_mouse_event()的代码中，我们看到，这个函数一方面通过 wake_up_interruptible()唤醒等待中的进程，同时还调用了一个函数 kill_fasync()：

==================== drivers/char/pc_keyb.c 396 397 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()]

```
static inline void handle_mouse_event(unsigned char scancode)
{
 ......
```

==================== drivers/char/pc_keyb.c 420 420 ====================

```
			kill_fasync(&queue->fasync, SIGIO, POLL_IN);
 ......
```

==================== drivers/char/pc_keyb.c 425 425 ====================

```
}
```

函数 kill_fasync()扫描鼠标器的 fasync 队列，向每个有关的进程发出一个 SIGIO 信号，并将 POLL_IN 传给各个进程的 SIGIO 信号服务程序作为参数，使其知道接收到信号的原因是通道中有了输入。函数 kill_fasync()的代码在 fs/fcntl.c 中：

==================== fs/fcntl.c 501 506 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()>kill_fasync()]

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
{
	read_lock(&fasync_lock);
	__kill_fasync(*fp, sig, band);
	read_unlock(&fasync_lock);
}
```

函数__kill_fasync()也在同一文件中：

==================== fs/fcntl.c 482 499 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()>kill_fasync()>__kill_fasync()]

```
void __kill_fasync(struct fasync_struct *fa, int sig, int band)
{
	while (fa) {
		struct fown_struct * fown;
		if (fa->magic != FASYNC_MAGIC) {
			printk(KERN_ERR "kill_fasync: bad magic number in "
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    "fasync_struct!\n");
                return;
            }
            fown = &fa->fa_file->f_owner;
            /* Don't send SIGURG to processes which have not set a
               queued signum: SIGURG has its own default signalling
               mechanism. */
            if (fown->pid && !(sig == SIGURG && fown->signum == 0))
                send_sigio(fown, fa->fa_fd, band);
            fa = fa->fa_next;
        }
}
```

对于队列中的每一个 fasync_struct 结构，通过与其相联系的 file 结构便可以找到通道的“主人”，这就是需要发送信号的对象。函数 send_sigio()的代码在 fs/fcntl.c 中：

==================== fs/fcntl.c 413 433 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()>kill_fasync()>__kill_fasync()>send_sigio()]

```
void send_sigio(struct fown_struct *fown, int fd, int band)
{
        struct task_struct * p;
        int   pid = fown->pid;

        read_lock(&tasklist_lock);
        if ( (pid > 0) && (p = find_task_by_pid(pid)) ) {
            send_sigio_to_task(p, fown, fd, band);
            goto out;
        }
        for_each_task(p) {
            int match = p->pid;
            if (pid < 0)
                match = -p->pgrp;
            if (pid != match)
                continue;
            send_sigio_to_task(p, fown, fd, band);
        }
out:
        read_unlock(&tasklist_lock);
}
```

如果根据对象的 pid 找到了这个进程，就只向这个进程发出信号，否则就向同一进程组中的所有进程都发出信号。函数 send_sigio_to_task()的代码也在同一文件（fs/fcntl.c）中：

==================== fs/fcntl.c 374 411 ====================
[keyboard_interrupt()>handle_kbd_event()>handle_mouse_event()>kill_fasync()>__kill_fasync()>send_sigio()>send_sigio_to_task()]

```
static void send_sigio_to_task(struct task_struct *p,
                    struct fown_struct *fown,
                    int fd,
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                    int reason)
{
    if ((fown->euid != 0) &&
       (fown->euid ^ p->suid) && (fown->euid ^ p->uid) &&
       (fown->uid ^ p->suid) && (fown->uid ^ p->uid))
        return;
    switch (fown->signum) {
        siginfo_t si;
        default:
            /* Queue a rt signal with the appropriate fd as its
              value.  We use SI_SIGIO as the source, not
              SI_KERNEL, since kernel signals always get
              delivered even if we can't queue.  Failure to
              queue in this case _should_ be reported; we fall
              back to SIGIO in that case. --sct */
            si.si_signo = fown->signum;
            si.si_errno = 0;
              si.si_code  = reason & ~__SI_MASK;
            /* Make sure we are called with one of the POLL_*
              reasons, otherwise we could leak kernel stack into
              userspace.  */
            if ((reason & __SI_MASK) != __SI_POLL)
                BUG();
            if (reason - POLL_IN >= NSIGPOLL)
                si.si_band  = ~0L;
            else
                si.si_band = band_table[reason - POLL_IN];
            si.si_fd    = fd;
            if (!send_sig_info(fown->signum, &si, p))
                break;
        /* fall-through: fall back on the old plain SIGIO signal */
        case 0:
            send_sig(SIGIO, p, 1);
    }
}
```

如果没有对发送的信号另加指定，则 fown->signum 为 0，此时发送的信号为 SIGIO，通过 send_sig()发送。如果另加指定则为某种实时信号，通过 send_sig_info()发送。我们把这段代码留给读者，作为对信号机制的一次复列。

8.11　设备文件系统 devfs

我们以前多次讲到过，以主设备号 / 次设备号为基础的设备文件管理方式是有根本性的缺点的。这种从 Unix 早期一直延用下来的方案一方面给设备号的管理带来了麻烦，一方面也破坏了/dev 目录的结构。Unix/Linux 系统中所有目录的结构都是层次的，惟独/dev 目录是“平面”的。这不光是风格的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

问题，也直接影响着访问的效率和管理的方便与否。而且，/dev 目录下的节点并不是按实际的需要而创建的，目录中存在某种设备的节点并不说明内核中有这种设备的驱动程序，更不说明系统中实际连接着这种设备。事实上，几乎所有 Unix/Linux 系统的/dev 目录下都有着大量实际不用的节点。这些节点的存在既降低了效率，又给管理带来麻烦，但是一般又不敢把它们删去，因为一来不知道究竟哪些是要直接或间接用到的，二来也不知道会不会将来某一天突然需要用到其中的某些节点了。究其原因，问题在于/dev 目录中的节点都是在内核的外部创建的，与内核的构成和运行并无直接的联系。

那么，理想的/dev 目录应该是什么样的呢？首先，它应该是层次的、树状的（不一定是严格意义上的树）。其次，它的规模应该是可伸缩的，而且不受数量的限制（例如 256 个主设备号）。还有，/dev 目录中的内容应该反映系统当前在设备驱动方面的实际情形。例如，这样一套方案就是比较理想的：

(1) 系统加电之初/dev 目录为空。
(2) 系统在初始化阶段扫描并枚举所有连接着的设备，就像对 PCI 总线的扫描枚举一样。每找到一项设备就分门别类地在/dev 目录下创建起子目录，然后以设备的序号作为最底层的节点名，例如“ /dev/ide/hd/1”、“ /dev/ide/floppy/1”等等。
(3) 以后，每插入一个设备，或安装一个可安装模块，就由内核在/dev 子树中增加一个或几个节点。
(4) 反之，如果关闭或拆除一个设备，或拆除一个可安装模块，就由内核从/dev 子树中删去相应的节点。
(5) 还得与原来的方案兼容。

除将/dev 目录改成树状以外，这里的关键在于将其纳入内核的管理，而不是像以前那样从内核外部管理，那正是造成不一致的原因。如果我们回顾一下前几章中的内容，就可以想起 Linux 其实有一个与此相似的特殊文件系统，那就是/proc。虽然管理的对象和目的不尽相同，但是在方法上显然是可以借鉴的。特殊文件系统 devfs 正是为实现上述目标而设计、与/proc 很相似的“文件系统”。

目前，devfs 的使用还只是一个实验性的选择项，由一个编译选择项 CONFIG_DEVFS_FS 加以选择。

文件系统类型 devfs_fs_type 的定义见 fs/devfs/base.c：

==================== fs/devfs/base.c 3145 3145 ====================
```
3145 static DECLARE_FSTYPE (devfs_fs_type, DEVFS_NAME, devfs_read_super, FS_SINGLE);
```

经过 gcc 的编译预处理以后，就会成为如下的定义：

```
struct file_system_type devfs_fs_type = {
        name:       "devfs",
        read_super: devfs_read_super,
        fs_flags:   FS_SINGLE,
        owner:      THIS_MODULE,
}
```

系统在初始化时会调用 init_devfs_fs()进行对 devfs 特殊文件系统的初始化，这个函数的代码在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 3342 3361 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int __init init_devfs_fs (void)
{
    int err;

    printk ("%s: v%s Richard Gooch (rgooch@atnf.csiro.au)\n",
         DEVFS_NAME, DEVFS_VERSION);
#ifdef CONFIG_DEVFS_DEBUG
    devfs_debug = devfs_debug_init;
    printk ("%s: devfs_debug: 0x%0x\n", DEVFS_NAME, devfs_debug);
#endif
    printk ("%s: boot_options: 0x%0x\n", DEVFS_NAME, boot_options);
    err = register_filesystem (&devfs_fs_type);
    if (!err)
    {
     struct vfsmount *devfs_mnt = kern_mount (&devfs_fs_type);
     err = PTR_ERR (devfs_mnt);
     if ( !IS_ERR (devfs_mnt) ) err = 0;
    }
    return err;
}   /*  End Function init_devfs_fs  */
```

首先通过 register_filesystem()向系统登记文件系统类型 devfs_fs_type，读者对这个函数应该已经比较熟悉了。然后，就通过 kern_mount()初始安装特殊文件系统 devfs。读者在第 5 章中看到，通过 kern_mount()安装的子系统其实并没有纳入以“/”为根的文件系统中，暂时还游离在外面。这样的文件系统还要再经过一次常规的安装才能纳入到以“/”为根的文件系统中，与其中的某个节点挂上钩，数据结构 devfs_fs_type 的字段 fs_flags 中标志位 FS_SINGLE 为 1 也说明了这一点。事实上，此刻的节点“/”还是空的，对 devfs 的初始安装先于根设备的安装。之所以要在安装根设备之前先初始安装 devfs，是因为在引导系统时可以通过一个命令行选择项指定以哪一个设备作为系统的根设备，所以在安装根设备之前就得要有最低限度的根据设备路径名找到其设备号的功能。

安装的本来意义是将一个设备上的文件系统子树跟已经存在于文件系统中的一个节点挂钩。可是，特殊文件系统实际上并不存在于某个设备上，所以要为其设置一个虚拟的设备。这个设备的主设备号为 UNNAMED_MAJOR，次设备号则由一个函数 get_unnamed_dev()分配。读者已经在第 5 章中结合/proc 的安装读过 kern_mount()的代码，知道在初始安装特殊文件系统时要调用一个函数 read_super()，而 read_super()，则通过具体 file_system_type 结构中的函数指针 read_super 读入或生成其超级块。从上面数据结构 devfs_fs_type 的定义中可以看到，devfs 的 read_super 操作是 devfs_read_super()，这个函数的代码在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 3112 3142 ====================

[init_devfs_fs()>kern_mount()>read_super()>devfs_read_super()]

```
static struct super_block *devfs_read_super (struct super_block *sb,
                               void *data, int silent)
{
    struct inode *root_inode = NULL;

    if (get_root_entry () == NULL) goto out_no_root;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    atomic_set (&fs_info.devfsd_overrun_count, 0);
    init_waitqueue_head (&fs_info.devfsd_wait_queue);
    init_waitqueue_head (&fs_info.revalidate_wait_queue);
    fs_info.sb = sb;
    sb->u.generic_sbp = &fs_info;
    sb->s_blocksize = 1024;
    sb->s_blocksize_bits = 10;
    sb->s_magic = DEVFS_SUPER_MAGIC;
    sb->s_op = &devfs_sops;
    if ( ( root_inode = get_vfs_inode (sb, root_entry, NULL) ) == NULL )
     goto out_no_root;
    sb->s_root = d_alloc_root (root_inode);
    if (!sb->s_root) goto out_no_root;
#ifdef CONFIG_DEVFS_DEBUG
    if (devfs_debug & DEBUG_DISABLED)
     printk ("%s: read super, made devfs ptr: %p\n",
          DEVFS_NAME, sb->u.generic_sbp);
#endif
    return sb;

out_no_root:
    printk ("devfs_read_super: get root inode failed\n");
    if (root_inode) iput (root_inode);
    return NULL;
}   /*  End Function devfs_read_super  */
```

像/proc 一样，devfs 在磁盘上也没有对应物，不像常规的文件系统那样在磁盘上有连接成树状的目录节点和索引节点，所以在内存中要为之建立起一套连接成树状的数据结构。对于 devfs 文件系统，这种数据结构是 devfs_entry，定义于 fs/devfs/base.c：

==================== fs/devfs/base.c 630 654 ====================

```
struct devfs_entry
{
    void *info;
    union
    {
     struct directory_type dir;
     struct fcb_type fcb;
     struct symlink_type symlink;
     struct fifo_type fifo;
    }
    u;
    struct devfs_entry *prev;    /*  Previous entry in the parent directory  */
    struct devfs_entry *next;    /*  Next entry in the parent directory      */
    struct devfs_entry *parent;  /*  The parent directory                    */
    struct devfs_entry *slave;   /*  Another entry to unregister             */
    struct devfs_inode inode;
    umode_t mode;
    unsigned short namelen;  /*  I think 64k+ filenames are a way off...  */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    unsigned char registered:1;
    unsigned char show_unreg:1;
    unsigned char hide:1;
    unsigned char no_persistence:1;
    char name[1];          /*  This is just a dummy: the allocated array is
                     bigger. This is NULL-terminated  */
};
```

结构中的字符数组 name[]就是节点名，其大小在为具体的数据结构分配空间时根据具体的字符串长度确定。每个 devfs_entry 结构通过指针 prev、next、parent 和 slave 连接成树状，以此来实现一个文件系统子树。从数据结构的定义可以看出，devfs 子树中的节点有四种不同的类型。第一种是 dir，即目录，这是不言自明的。第二种是 fcb，即“文件控制块”，这是 devfs 子树中的叶节点。第三种是 symlink、这也是不言自明的。最后一种是 fifo，专门用于管道文件。因节点类型的不同，devfs_entry 结构中的 union_u 也相应地解释成不同的数据结构。这里先看一 fcb_type 数据结构的定义（fs/devfs/base.c）：

==================== fs/devfs/base.c 577 603 ====================

```
struct file_type
{
    unsigned long size;
};

struct device_type
{
    unsigned short major;
    unsigned short minor;
};

struct fcb_type  /*  File, char, block type  */
{
    uid_t default_uid;
    gid_t default_gid;
    void *ops;
    union
    {
      struct file_type file;
      struct device_type device;
    }
    u;
    unsigned char auto_owner:1;
    unsigned char aopen_notify:1;
    unsigned char removable:1;  /*  Belongs in device_type, but save space   */
    unsigned char open:1;      /*  Not entirely correct                  */
};
```

可见，devfs 中的叶节点有两种类型，一种是文件，另一种是设备。对于设备，fcb_type 结构提供了 16 位的主 / 次设备号。不过，如前所述，在 devfs 中主设备号与具体的驱动程序没有固定的对应关系，而是动态分配的。每个 file_type 数据结构中都有个指针 ops，根据需要指向具体的 file_operations

PDF-XCHANGE Click to buy NOW! www.docu-track.com

或其它数据结构。

内核中还有个数据结构 fs_info，也定义于 fs/devfs/base.c：

==================== fs/devfs/base.c 668 685 ====================

```
struct fs_info     /*  This structure is for each mounted devfs  */
{
    unsigned int num_inodes;    /*  Number of inodes created        */
    unsigned int table_size;    /*  Size of the inode pointer table  */
    struct devfs_entry **table;
    struct super_block *sb;
    volatile struct devfsd_buf_entry *devfsd_buffer;
    volatile unsigned int devfsd_buf_in;
    volatile unsigned int devfsd_buf_out;
    volatile int devfsd_sleeping;
    volatile int devfsd_buffer_in_use;
    volatile struct task_struct *devfsd_task;
    volatile struct file *devfsd_file;
    volatile unsigned long devfsd_event_mask;
    atomic_t devfsd_overrun_count;
    wait_queue_head_t devfsd_wait_queue;
    wait_queue_head_t revalidate_wait_queue;
};
```

这个数据结构中的指针 table 指向一个 devfs_entry 指针数组，table_size 为数组当前的大小。这个数组中的指针分别指向所有已经分配的 devfs_entry 数据结构，或者说 devfs 子树中当前的所有节点。一开始时内核中没有已经分配的 devfs_entry 数据结构，所以这个数组的大小为 0。

内核在设备驱动程序中仍然使用设备号，不过主 / 次设备号都已改成 16 位，并从中划出一块供 devfs 动态分配。原来已经分配了主 / 次设备号的还继续延用原来的设备号，而未经分配的则可以在初始化阶段向 devfs 登记时让 devfs 动态地分配一个。动态分配的设备号只是临时的，其作用仅在于在 devfs 中的节点与数组 chrdevs[]或 blk_dev[]中的元素建立起联系。这样，只要应用软件按预定的路径名打开设备文件，就能得到目标设备当前的设备号。

对于字符设备与块设备，供 devfs 动态分配的设备号都从 MIN_DEVNUM 开始（fs/devfs/base.c）：

==================== fs/devfs/base.c 527 528 ====================

```
#define MIN_DEVNUM 36864  /*  Use major numbers 144   */
#define MAX_DEVNUM 61439  /*  through 239, inclusive  */
```

==================== fs/devfs/base.c 688 689 ====================

```
static unsigned int next_devnum_char = MIN_DEVNUM;
static unsigned int next_devnum_block = MIN_DEVNUM;
```

第一个 devfs_entry 数据结构当然应该是 devfs 的根节点，函数 get_root_entry()的代码在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 841 870 ====================

[init_devfs_fs()>kern_mount()>read_super()>devfs_read_super()>get_root_entry()]

```
/**
 *   get_root_entry - Get the root devfs entry.
```

```
 *
 *   Returns the root devfs entry on success, else %NULL.
 */

static struct devfs_entry *get_root_entry (void)
{
    struct devfs_entry *new;

    /*  Always ensure the root is created  */
    if (root_entry != NULL) return root_entry;
    if ( ( root_entry = create_entry (NULL, NULL, 0) ) == NULL ) return NULL;
    root_entry->registered = TRUE;
    root_entry->mode = S_IFDIR;
    /*  Force an inode update, because lookup() is never done for the root  */
    update_devfs_inode_from_entry (root_entry);
    /*  And create the entry for ".devfsd"  */
    if ( ( new = create_entry (root_entry, ".devfsd", 0) ) == NULL )
      return NULL;
    new->registered = TRUE;
    new->u.fcb.u.device.major = next_devnum_char >> 8;
    new->u.fcb.u.device.minor = next_devnum_char & 0xff;
    ++next_devnum_char;
    new->mode = S_IFCHR | S_IRUSR | S_IWUSR;
    new->u.fcb.default_uid = 0;
    new->u.fcb.default_gid = 0;
    new->u.fcb.ops = &devfsd_fops;
    return root_entry;
}   /*  End Function get_root_entry  */
```

先创建 devfs 的根节点 root_entry，这里指针 root_entry 是个全局量。

==================== fs/devfs/base.c 764 808 ====================

[init_devfs_fs()>kern_mount()>read_super()>devfs_read_super()>get_root_entry()>create_entry()]

```
static struct devfs_entry *create_entry (struct devfs_entry *parent,
                                const char *name,unsigned int namelen)
{
    struct devfs_entry *new, **table;

    /*  First ensure table size is enough  */
    if (fs_info.num_inodes >= fs_info.table_size)
    {
      if ( ( table = kmalloc (sizeof *table *
                     (fs_info.table_size + INODE_TABLE_INC),
                     GFP_KERNEL) ) == NULL ) return NULL;
      fs_info.table_size += INODE_TABLE_INC;
#ifdef CONFIG_DEVFS_DEBUG
      if (devfs_debug & DEBUG_I_CREATE)
         printk ("%s: create_entry(): grew inode table to: %u entries\n",
              DEVFS_NAME, fs_info.table_size);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
780 #endif
781          if (fs_info.table)
782          {
783              memcpy (table, fs_info.table, sizeof *table *fs_info.num_inodes);
784              kfree (fs_info.table);
785          }
786          fs_info.table = table;
787      }
788      if ( name && (namelen < 1) ) namelen = strlen (name);
789      if ( ( new = kmalloc (sizeof *new + namelen, GFP_KERNEL) ) == NULL )
790          return NULL;
791      /*  Magic: this will set the ctime to zero, thus subsequent lookups will
792          trigger the call to <update_devfs_inode_from_entry>  */
793      memset (new, 0, sizeof *new + namelen);
794      new->parent = parent;
795      if (name) memcpy (new->name, name, namelen);
796      new->namelen = namelen;
797      new->inode.ino = fs_info.num_inodes + FIRST_INODE;
798      new->inode.nlink = 1;
799      fs_info.table[fs_info.num_inodes] = new;
800      ++fs_info.num_inodes;
801      if (parent == NULL) return new;
802      new->prev = parent->u.dir.last;
803      /*  Insert into the parent directory's list of children  */
804      if (parent->u.dir.first == NULL) parent->u.dir.first = new;
805      else parent->u.dir.last->next = new;
806      parent->u.dir.last = new;
807      return new;
808 }   /*  End Function create_entry  */
```

当 fs_info 数据结构中的 num_inodes 字段和 table_size 字段相等时，就需要扩充由 fs_info.table 所指的 devfs_entry 指针数组了。每次扩充都在原来的大小上增加 INODE_TABLE_INC，即 250 个指针的容量。显然，创建 devfs 的根节点时这两个字段都是 0，所以要为最初的 250 个指针分配空间。如果不是第一次分配空间，则还要把已经存在的数组复制到新的空间中，并释放原有的空间。接着，就要为 devfs_entry 数据结构本身(连同节点名字符串)分配空间。用于 devfs 根节点的 inode 号为 FIRST_INODE，即 1，以后则逐次递增。除根节点外，devfs 文件系统中的每个节点都有个父节点，而其父节点必定是个目录节点，802～806 行将新的节点与父节点以及同一目录中的其它节点链接在一起。

回到 get_root_entry()中，下一步要在 devfs 的根节点下创建一个节点“.devfsd”。这个节点是个 fcb 节点。在 devfs 中，设备号是由系统自动分配的，设备号起点是((((? ? ?)))读者将看到，设备号在 devfs 中并不起着原来那么重要的作用。此外，fcb 节点都有个指针 ops 指向一个 file_operations 数据结构。对于节点“.devfsd”，这个数据结构是定义于 fs/devfs/base.c 的 devfsd_fops：

==================== fs/devfs/base.c 715 721 ====================

```
/*  Devfsd daemon file operations  */
static struct file_operations devfsd_fops =
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    read:    devfsd_read,
    ioctl:   devfsd_ioctl,
    release: devfsd_close,
};
```

这是个特殊的节点，这个节点并不是代表着一项具体的设备，而是反映着 devfs 当前的内容变化，其意图是在用户空间创建一个 devfs 的“保护神”进程 devfsd。这个进程的主体是个无穷循环，并总是因为读/dev/.devfsd 受阻而睡眠。但是，每当 devfs 的内容发生变化时，例如创建或删除一个节点或打开一个设备文件时，内核就（通过一个叫 devfsd_notify()的函数）将这个进程唤醒。这样，就可以安排这个进程每当 devfs 的内容发生变化时就作出一些反应，例如在屏幕上显示一行信息“ /dev/x/y/z/1 is up”。

同到 devfs_read_super()的代码中，下面就是填写 devfs 的 super_block 数据结构的内容，注意这里把指针 s_op 设置成指向 devfs_sops，定义于 fs/devfs/base.c：

==================== fs/devfs/base.c 2364 2369 ====================

```
static struct super_operations devfs_sops =
{
    read_inode:    devfs_read_inode,
    write_inode:   devfs_write_inode,
    statfs:        devfs_statfs,
};
```

然后就通过 get_vfs_inode()为 devfs 的根节点创建一个 inode 数据结构（fs/devfs/base.c）：

==================== fs/devfs/base.c 2381 2399 ====================
[init_devfs_fs()>kern_mount()>read_super()>devfs_read_super()>get_vfs_inode()]

```
static struct inode *get_vfs_inode (struct super_block *sb,
                         struct devfs_entry *de,
                         struct dentry *dentry)
{
    struct inode *inode;

    if (de->inode.dentry != NULL)
    {
     printk ("%s: get_vfs_inode(%u): old de->inode.dentry: %p \"%s\"  new dentry: %p \"%s\"\n",
          DEVFS_NAME, de->inode.ino,
          de->inode.dentry, de->inode.dentry->d_name.name,
          dentry, dentry->d_name.name);
     printk ("  old inode: %p\n", de->inode.dentry->d_inode);
     return NULL;
    }
    if ( ( inode = iget (sb, de->inode.ino) ) == NULL ) return NULL;
    de->inode.dentry = dentry;
    return inode;
}  /*  End Function get_vfs_inode  */
```

读者在第 5 章中已经看到，iget()首先在 inode 结构的杂凑队列中查找，如果找不到就新创建一个。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

最后，通过 d_alloc_root()为 devfs 的根节点创建一个 dentry 数据结构，这个节点也叫“ /”，但只是 devfs 的根，与整个文件系统的总根不是一回事。

完成了对超级块的处理以后，kern_mount()将 devfs 的根节点暂时“安装”到了 devfs 的 file_system_type 数据结构 devfs_fs_type 上，使其指针 kern_mnt 指向用作“连接件”的 vfsmount 结构，为以后进一步安装作好了准备。所以，这种初始的安装也可以称作“预安装”，有关详情可参阅/proc 文件系统的安装。

在安装了系统的总根以后，系统的初始化过程还会调用 mount_devfs_fs()对 devfs 作进一步的安装（fs/devfs/base.c）：

==================== fs/devfs/base.c 3361 3371 ====================

```
void __init mount_devfs_fs (void)
{
    int err;

    if ( (boot_options & OPTION_NOMOUNT) ) return;
    err = do_mount ("none", "/dev", "devfs", 0, "");
    if (err == 0) printk ("Mounted devfs on /dev\n");
    else printk ("Warning: unable to mount devfs, err: %d\n", err);
}   /*  End Function mount_devfs_fs  */
```

可见，特殊文件系统 devfs 的安装点是“ /dev”。由于 devfs_fs_type 中的 FS_SINGLE 标志位为 1，安装时会把保存在 devfs_fs_type 中的 vfsmount 结构指针安装到“ /dev”节点上。

完成了 devfs 的安装以后，各种设备的驱动程序就可以通过 devfs_register_chrdev()或 devfs_register_blkdev()向 devfs 登记，从而在/dev 目录下创建起相应的节点。

(1) 通过 devfs_register_chrdev()向 devfs 登记一类设备的主设备号、设备名以及 file_operations 数据结构，建立起三者间的联系。主设备号可以是静态指定的，也可以要求 devfs 动态地分配。
(2) 或者通过 devfs_register_blkdev()向 devfs 登记一类设备的主设备号、设备名以及 block_device_operations 数据结构，建立起三者间的联系。同样，主设备号可以是静态指定的，也可以要求 devfs 动态地分配。
(3) 通过 devfs_mk_dir()建立目录节点。
(4) 通过 devfs_register()登记具体的设备，并在指定目录下建立叶节点。

在“可安装模块”一节所引的实例中，读者可以看到一个声卡驱动模块在初始化时首先通过 devfs_register_chrdev()登记驱动程序，并且看到这个函数实际上通过 register_chrdev()完成登记。所谓“登记”，实际上就是把给定的 file_operations 结构指针填入 chrdevs[]的某个元素，而设备的主设备号，则就是用于这个数组的下标。在老的方案中，主设备与都是静态地指定好的，声卡设备的主设备号就是 SOUND_MAJOR。而在 devfs 中，则既可以继续延用静态的主设备号，也可以在调用 devfs_register_chrdev()或 register_chrdev()时以 0 为主设备号，表示要求 devfs 动态地分配一个。其实，不管是否采用 devfs，都要先进行登记，这样才能在打开文件时根据主设备号找到该设备的 file_operations 结构。不同的是，在老方案中这个主设备号必须是预知的，这样才能从内核外部通过 mknod()在/dev 目录下创建起相应的节点。而在 devfs 中，则可以先动态分配主设备号（并登记），然后

才在内核中（内部）通过 devfs_register()在特殊文件系统中创建节点。而且，在 devfs_register()之前还可以根据需要通过 devfs_mk_dir()建立起若干中间节点。这样，只要在设备驱动模块与应用程序在路径名的使用上有默契就行了，主设备号的使用实际上是透明的。所以，devfs_mk_dir()和 devfs_register()才是 devfs 的关键所在。先看 devfs_mk_dir()，其代码在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 1530 1586 ====================

```
1530 /**
1531  *   devfs_mk_dir - Create a directory in the devfs namespace.
1532  *   @dir: The handle to the parent devfs directory entry. If this is %NULL the
1533  *        new name is relative to the root of the devfs.
1534  *   @name: The name of the entry.
1535  *   @info: An arbitrary pointer which will be associated with the entry.
1536  *
1537  *   Use of this function is optional. The devfs_register() function
1538  *   will automatically create intermediate directories as needed. This function
1539  *   is provided for efficiency reasons, as it provides a handle to a directory.
1540  *   Returns a handle which may later be used in a call to devfs_unregister().
1541  *   On failure %NULL is returned.
1542  */
1543
1544 devfs_handle_t devfs_mk_dir (devfs_handle_t dir, const char *name, void *info)
1545 {
1546     int is_new;
1547     struct devfs_entry *de;
1548
1549     if (name == NULL)
1550     {
1551      printk ("%s: devfs_mk_dir(): NULL name pointer\n", DEVFS_NAME);
1552      return NULL;
1553     }
1554     de = search_for_entry (dir, name, strlen (name), TRUE, TRUE, &is_new,
1555                   FALSE);
1556     if (de == NULL)
1557     {
1558      printk ("%s: devfs_mk_dir(): could not create entry: \"%s\"\n",
1559           DEVFS_NAME, name);
1560      return NULL;
1561     }
1562     if (!S_ISDIR (de->mode) && de->registered)
1563     {
1564      printk ("%s: devfs_mk_dir(): existing non-directory entry: \"%s\"\n",
1565           DEVFS_NAME, name);
1566      return NULL;
1567     }
1568 #ifdef CONFIG_DEVFS_DEBUG
1569     if (devfs_debug & DEBUG_REGISTER)
1570      printk ("%s: devfs_mk_dir(%s): de: %p %s\n",
1571           DEVFS_NAME, name, de, is_new ? "new" : "existing");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#endif
    if (!S_ISDIR (de->mode) && !is_new)
    {
      /*  Transmogrifying an old entry  */
      de->u.dir.first = NULL;
      de->u.dir.last = NULL;
    }
    de->mode = S_IFDIR | S_IRUGO | S_IXUGO;
    de->info = info;
    if (!de->registered) de->u.dir.num_removable = 0;
    de->registered = TRUE;
    de->show_unreg = (boot_options & OPTION_SHOW) ? TRUE : FALSE;
    de->hide = FALSE;
    return de;
}  /*  End Function devfs_mk_dir  */
```

参数 dir 指向父目录的 devfs_entry 结构，数据类型 devfs_handle_t 实际上就是 devfs_entry 指针，定义于 include/linux/devfs_fs_kernel.h：

==================== include/linux/devfs_fs_kernel.h 45 45 ====================

```
typedef struct devfs_entry * devfs_handle_t;
```

这个函数的主体是 search_for_entry()，通过它在 dir 下面的子树中找到或者创建目标节点的 devfs_entry 结构。然后，如果是新创的节点，则加以设置。函数 search_for_entry()的代码也在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 873 952 ====================

[devfs_mk_dir()>search_for_entry()]

```
/**
 *   search_for_entry - Search for an entry in the devfs tree.
 *   @dir: The parent directory to search from. If this is %NULL the root is used
 *   @name: The name of the entry.
 *   @namelen: The number of characters in @name.
 *   @mkdir: If %TRUE intermediate directories are created as needed.
 *   @mkfile: If %TRUE the file entry is created if it doesn't exist.
 *   @is_new: If the returned entry was newly made, %TRUE is written here. If
 *        this is %NULL nothing is written here.
 *   @traverse_symlink: If %TRUE then symbolic links are traversed.
 *
 *   If the entry is created, then it will be in the unregistered state.
 *   Returns a pointer to the entry on success, else %NULL.
 */

static struct devfs_entry *search_for_entry (struct devfs_entry *dir,
                                             const char *name,
                                             unsigned int namelen, int mkdir,
                                             int mkfile, int *is_new,
                                             int traverse_symlink)
{
```

```
    int len;
    const char *subname, *stop, *ptr;
    struct devfs_entry *entry;

    if (is_new) *is_new = FALSE;
    if (dir == NULL) dir = get_root_entry ();
    if (dir == NULL) return NULL;
    /*  Extract one filename component  */
    subname = name;
    stop = name + namelen;
    while (subname < stop)
    {
      /*  Search for a possible '/'  */
      for (ptr = subname; (ptr < stop) && (*ptr != '/'); ++ptr);
      if (ptr >= stop)
      {
          /*  Look for trailing component  */
          len = stop - subname;
          entry = search_for_entry_in_dir (dir, subname, len,
                                           traverse_symlink);
          if (entry != NULL) return entry;
          if (!mkfile) return NULL;
          entry = create_entry (dir, subname, len);
          if (entry && is_new) *is_new = TRUE;
          return entry;
      }
      /*  Found '/': search for directory  */
      if (strncmp (subname, "../", 3) == 0)
      {
          /*  Going up  */
          dir = dir->parent;
          if (dir == NULL) return NULL;  /*  Cannot escape from devfs  */
          subname += 3;
          continue;
      }
      len = ptr - subname;
      entry = search_for_entry_in_dir (dir, subname, len, traverse_symlink);
      if (!entry && !mkdir) return NULL;
      if (entry == NULL)
      {
          /*  Make it  */
          if ( ( entry = create_entry (dir, subname, len) ) == NULL )
            return NULL;
          entry->mode = S_IFDIR | S_IRUGO | S_IXUGO | S_IWUSR;
          if (is_new) *is_new = TRUE;
      }
      if ( !S_ISDIR (entry->mode) )
      {
          printk ("%s: existing non-directory entry\n", DEVFS_NAME);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
943          return NULL;
944       }
945       /*  Ensure an unregistered entry is re-registered and visible  */
946       entry->registered = TRUE;
947       entry->hide = FALSE;
948       subname = ptr + 1;
949       dir = entry;
950     }
951     return NULL;
952 }  /*  End Function search_for_entry  */
```

这个函数返回指向找到或新创建的 devfs_entry 结构的指针，并通过副作用返回参数 is_new，表明这个 devfs_entry 结构是否新创。如果参数 dir 为 0，则以 devfs 的根节点为父节点。参数 name 指向一个相对路径名。代码中通过一个 while 循环顺着相对路径名中的节点逐步向前推进，每一步都通过 search_for_entry_in_dir()在当前目录中找到下一个节点（930 行），如果找不到就通过 create_entry()自动补上一个中间节点。最后，当路径名中剩下的部分不再有“/”字符存在时，这就是目标节点了（908 行）。对于目标节点，也是先通过 search_for_entry_in_dir()在当前目录中寻找，如果找不到就通过 create_entry 创建。对于读过 path_walk()的读者，这只不过是“小菜一碟”。至于 search_for_entry_in_dir()，其代码也在 fs/devfs/base.c 中，我们把它列在下面供读者自己阅读，从中可以理解对 devfs_entry 结构中的 union 的运用。

==================== fs/devfs/base.c 738 762 ====================

[devfs_mk_dir()>search_for_entry()>search_for_entry_in_dir()]

```
static struct devfs_entry *search_for_entry_in_dir (struct devfs_entry *parent,
                                               const char *name,
                                               unsigned int namelen,
                                               int traverse_symlink)
{
    struct devfs_entry *curr;

    if ( !S_ISDIR (parent->mode) )
    {
      printk ("%s: entry is not a directory\n", DEVFS_NAME);
      return NULL;
    }
    for (curr = parent->u.dir.first; curr != NULL; curr = curr->next)
    {
      if (curr->namelen != namelen) continue;
      if (memcmp (curr->name, name, namelen) == 0) break;
      /*  Not found: try the next one  */
    }
    if (curr == NULL) return NULL;
    if (!S_ISLNK (curr->mode) || !traverse_symlink) return curr;
    /*  Need to follow the link: this is a stack chomper  */
    return search_for_entry (parent,
                         curr->u.symlink.linkname, curr->u.symlink.length,
                         FALSE, FALSE, NULL, TRUE);
```

```
762  }   /*  End Function search_for_entry_in_dir  */
```

通过 devfs_mk_dir()创建了所需的目录节点以后，就在 devfs 的目录系统中为目标设备建立起了一个框架，最后还要通过 devfs_register()为具体的设备创建目标节点。这个函数的代码在 fs/devfs/base.c 中：

==================== fs/devfs/base.c 1214 1366 ====================

```
1214 /**
1215  *   devfs_register - Register a device entry.
1216  *   @dir: The handle to the parent devfs directory entry. If this is %NULL the
1217  *        new name is relative to the root of the devfs.
1218  *   @name: The name of the entry.
1219  *   @flags: A set of bitwise-ORed flags (DEVFS_FL_*).
1220  *   @major: The major number. Not needed for regular files.
1221  *   @minor: The minor number. Not needed for regular files.
1222  *   @mode: The default file mode.
1223  *   @ops: The &file_operations or &block_device_operations structure.
1224  *        This must not be externally deallocated.
1225  *   @info: An arbitrary pointer which will be written to the @private_data
1226  *        field of the &file structure passed to the device driver. You can set
1227  *        this to whatever you like, and change it once the file is opened (the next
1228  *        file opened will not see this change).
1229  *
1230  *   Returns a handle which may later be used in a call to devfs_unregister().
1231  *   On failure %NULL is returned.
1232  */
1233
1234 devfs_handle_t devfs_register (devfs_handle_t dir, const char *name,
1235                     unsigned int flags,
1236                     unsigned int major, unsigned int minor,
1237                     umode_t mode, void *ops, void *info)
1238 {
1239     int is_new;
1240     struct devfs_entry *de;
1241
1242     if (name == NULL)
1243     {
1244      printk ("%s: devfs_register(): NULL name pointer\n", DEVFS_NAME);
1245      return NULL;
1246     }
1247     if (ops == NULL)
1248     {
1249      if ( S_ISBLK (mode) ) ops = (void *) get_blkfops (major);
1250      if (ops == NULL)
1251      {
1252          printk ("%s: devfs_register(%s): NULL ops pointer\n",
1253               DEVFS_NAME, name);
1254          return NULL;
1255      }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     printk ("%s: devfs_register(%s): NULL ops, got %p from major table\n",
         DEVFS_NAME, name, ops);
    }
    if ( S_ISDIR (mode) )
    {
     printk("%s: devfs_register(%s): creating directories is not allowed\n",
          DEVFS_NAME, name);
     return NULL;
    }
    if ( S_ISLNK (mode) )
    {
     printk ("%s: devfs_register(%s): creating symlinks is not allowed\n",
         DEVFS_NAME, name);
     return NULL;
    }
    if ( S_ISCHR (mode) && (flags & DEVFS_FL_AUTO_DEVNUM) )
    {
     if (next_devnum_char >= MAX_DEVNUM)
     {
        printk ("%s: devfs_register(%s): exhausted char device numbers\n",
            DEVFS_NAME, name);
        return NULL;
     }
     major = next_devnum_char >> 8;
     minor = next_devnum_char & 0xff;
     ++next_devnum_char;
    }
    if ( S_ISBLK (mode) && (flags & DEVFS_FL_AUTO_DEVNUM) )
    {
     if (next_devnum_block >= MAX_DEVNUM)
     {
        printk ("%s: devfs_register(%s): exhausted block device numbers\n",
            DEVFS_NAME, name);
        return NULL;
     }
     major = next_devnum_block >> 8;
     minor = next_devnum_block & 0xff;
     ++next_devnum_block;
    }
    de = search_for_entry (dir, name, strlen (name), TRUE, TRUE, &is_new,
               FALSE);
    if (de == NULL)
    {
     printk ("%s: devfs_register(): could not create entry: \"%s\"\n",
         DEVFS_NAME, name);
     return NULL;
    }
#ifdef CONFIG_DEVFS_DEBUG
    if (devfs_debug & DEBUG_REGISTER)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     printk ("%s: devfs_register(%s): de: %p %s\n",
             DEVFS_NAME, name, de, is_new ? "new" : "existing");
#endif
    if (!is_new)
    {
     /*  Existing entry  */
     if ( !S_ISCHR (de->mode) && !S_ISBLK (de->mode) &&
          !S_ISREG (de->mode) )
     {
        printk ("%s: devfs_register(): existing non-device/file entry: \"%s\"\n",
             DEVFS_NAME, name);
        return NULL;
     }
     if (de->registered)
     {
        printk("%s: devfs_register(): device already registered: \"%s\"\n",
            DEVFS_NAME, name);
        return NULL;
     }
    }
    de->registered = TRUE;
    if ( S_ISCHR (mode) || S_ISBLK (mode) )
    {
     de->u.fcb.u.device.major = major;
     de->u.fcb.u.device.minor = minor;
    }
    else if ( S_ISREG (mode) ) de->u.fcb.u.file.size = 0;
    else
    {
     printk ("%s: devfs_register(): illegal mode: %x\n",
             DEVFS_NAME, mode);
     return (NULL);
    }
    de->info = info;
    de->mode = mode;
    if (flags & DEVFS_FL_CURRENT_OWNER)
    {
     de->u.fcb.default_uid = current->uid;
     de->u.fcb.default_gid = current->gid;
    }
    else
    {
     de->u.fcb.default_uid = 0;
     de->u.fcb.default_gid = 0;
    }
    de->registered = TRUE;
    de->u.fcb.ops = ops;
    de->u.fcb.auto_owner = (flags & DEVFS_FL_AUTO_OWNER) ? TRUE : FALSE;
    de->u.fcb.aopen_notify = (flags & DEVFS_FL_AOPEN_NOTIFY) ? TRUE : FALSE;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1354     if (flags & DEVFS_FL_REMOVABLE)
1355     {
1356      de->u.fcb.removable = TRUE;
1357      ++de->parent->u.dir.num_removable;
1358     }
1359     de->u.fcb.open = FALSE;
1360     de->show_unreg = ( (boot_options & OPTION_SHOW)
1361                 || (flags & DEVFS_FL_SHOW_UNREG) ) ? TRUE : FALSE;
1362     de->hide = (flags & DEVFS_FL_HIDE) ? TRUE : FALSE;
1363     de->no_persistence = (flags & DEVFS_FL_NO_PERSISTENCE) ? TRUE : FALSE;
1364     devfsd_notify (de, DEVFSD_NOTIFY_REGISTERED, flags & DEVFS_FL_WAIT);
1365     return de;
1366 }   /*  End Function devfs_register  */
```

我们知道，“设备文件”节点的实质就在于：把代表着目标设备的文件名（路径名）与具体的“对象”挂上钩，包括一套驱动程序（通过主设备号）以及具体的数据结构（通过次设备号）。在老的方案中，主设备号就惟一地确定了一个 file_operations 数据结构。相比之下，在 devfs 中提供了更大的灵活性，在调用 devfs_register()时可以通过参数 ops 传下一个 file_operations 结构指针，只是当 ops 为 0 时才采用由主设备号确定的 file_operations 数据结构。主 / 次设备号则既可以作为参数传下来，也可以由系统产生。

同样，这里也是通过 search_for_entry()在以 dir 为根的子树中找到或创建目标节点的 devfs_entry 数据结构（1295 行），只不过这一次把 devfs_entry 结构用作 fcb 节点，而不是目录节点。最后，这里还调用了一个函数 devfsd_notify()，目的在于为 devfsd 进程准备下一些信息并将其唤醒，让它知道现在 devfs 中又多了一个设备节点。

此后应用进程就可以通过新的 devfs 格式的路径名打开设备文件，此后的操作就与原来的没有什么不同了。显然，如果把目标节点直接建立在 devfs 的根目录，即/dev 下面，并且符合从前对设备名的约定，那么从应用程序的角度看就与老的方案没有什么不同了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

第9章 多处理器 SMP 系统结构

9.1 概述

在过去的 10 年里，微处理器的速度增长了近 100 倍，但是尽管如此，CPU 的最高速度总是受到当时技术条件的限制。在给定的时间里，CPU 能达到的最高速度总是给定的，再要提高速度就只好设法增加并行度，方法之一就是在计算机中使用多个处理器，所以，多处理器结构很长时间以来一直是计算机科学与技术的一个重要分支。在长期的研究中，人们先后开发了几种重要的多处理器系统结构模型，其中之一就是所谓“对称多处理器结构”(Semetric Multi-Processor Architecture)，缩写为 SMP，这是一种比较简单的多处理器系统结构。SMP 是一种系统结构的模型，不过为行文的方便我们在本书中也常常把采用 SMP 结构的计算机系统简称为 SMP 结构。还有，在本书中讲到 SMP 结构时大多专指采用 Intel Pentium 处理器的系统，但是也有些场合是泛指，读者要根据上下文加以区分。

在这种系统结构中，所有的 CPU 在运行时（除系统引导和初始化以外）都是“对称”的，或者说都是“平等”的，没有主次之分，通常物理上也采用同一种 CPU（在这样的系统中，实际上已经没有所谓“中央处理器”了，但为了行文方便我们仍使用 CPU 表示“处理器”）。所有的 CPU 通过同一条总线共享同一个内存以及所有的外设。为了减少访问内存的冲突，SMP 结构中的各个 CPU 通常都有自己的高速缓存。图 9.1 给出了 SMP 系统结构模型示意图。

对于 Intel 的 Pentium 处理器（以及高于 Pentium 的处理器，下同），系统中最多可以容纳 32 个平行的处理器。

各个 CPU 动态地从系统的就绪进程队列中调度（挑选出）进程加以执行。一个进程在不同的时间中可以在不同的 CPU 上运行；中断请求则动态地分配给其中的某个 CPU，由这个 CPU 提供中断服务。除一般的共享内存外，处理器间的通信手段还有进程间通信和处理器之间的中断请求。如前所述，通常每个 CPU 都配备了自己的高速缓冲存储器，以减少访问内存时的冲突；另一方面，虽然系统中所有的处理器都采用相同的时钟脉冲，但是由于指令的长短不一，加上访问内存时可能会有的冲突以及高速缓存的使用等等因素，一般向言指令的边界是不能对齐的。因此，系统中的各个 CPU 都在独立地、异步地执行指令。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

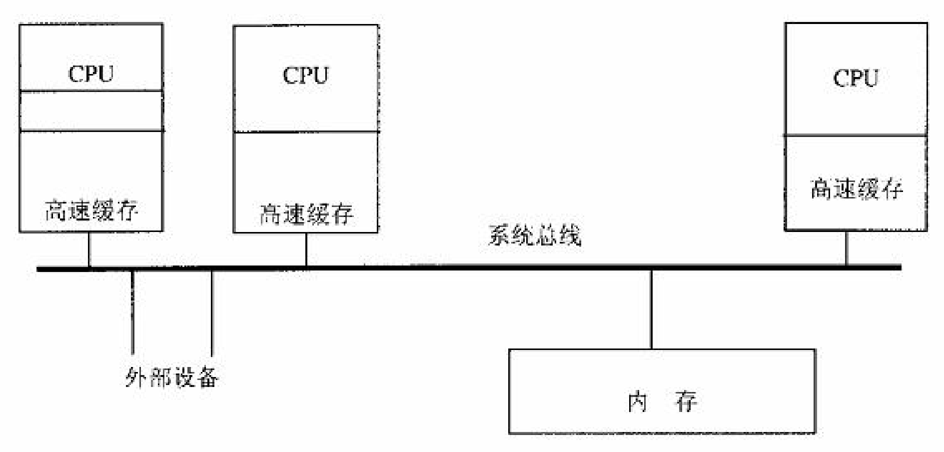


图 9.1 SMP 结构模型

这样的结构适合于要求提高系统总的“吞吐量”，但是系统中的各个进程则互相独立的那些应用。例如，网络服务器就是一个很好的例子，因为网络服务器在同一时间中要响应大量的“点击”，而对这些点击的服务又是互相独立的（尽管实际执行的程序很可能相同）。考虑到这一点，以及近年来对网络服务器的大量需求，就不难明白为什么这几年 SMP 结构变得这么热门了。至于是否可以通过采用 SMP 结构的计算机来提高解算大问题的进度，则取决于是否存在好的并行算法，从而可以有效地把给定的大问题分解成若干可以并发执行的进程。另一方面，SMP 是一种通用的系统结构，用它来实现一些重要的并行算法往往不如一些专用的系统结构（如“向量机”结构）那么有效。

但是，与单处理器结构相比，SMP 结构的实现有一些特殊的问题需要考虑和解决。

首先是处理器间的同步与互斥。从宏观上说这也是个进程间通信的问题，但是多个处理器的存在使这个问题更复杂化了。在单处理器结构中，各个进程在宏观上是并行的，但是在微观上却是串行的，因为在同一时间点上只有一个进程真正在运行（系统中只有一个处理器），因此称为“并发”。在这样的系统中，保证进程间的同步与互斥是比较容易的。回顾一下“临界区”的实现，就可以明白进程间的同步实际上可以归结到对临界资源的互斥操作。在单处理器结构中，只要能保证在对临界资源的操作中途不会发生进程调度，并且不会发生中断，或者即使发生了中断也与操作的对象无关，就保证了操作的互斥性。即使在极端的情况下（例如不允许关中断），只要对临界资源的操作能在单条指令中完成，那也保证了操作的互斥性，因为中断只能发生于指令之间，而不会发生在执行一条指令的中途。一般而言，只要能保证对临界资源操作的“原子性”，互斥性就保证了。所以，在单处理器结构中，能够在单条指令中完成的操作就认为是“原子操作”。这也是为什么一些 CPU 指令系统中设置了“测试并设置”、“测试并清除”等指令的原因，这些指令主要都是用于对临界资源的互斥操作。可是，在 SMP 结构中就不同了，由于系统中有多个处理器在独立地运行，即使能在单条指令中完成的操作也有可能受到干扰。就以“测试并设置”这条指令为例，它先从某个内存单元读出其内容，测试其中的某一位，并把这一位设置成 1（也可能原来就是 1），再写回内存单元中，并根据测试的结果设置标志位寄存器中的相应标志位。可见，在这条指令的执行过程中要访问内存两次，形战一个“读—改—写”的过程。这个过程中的每一步都是一个“微操作”，整条指令则由若干微操作构成。于是，在 SMP 结构中就完

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

全有可能发生这样的情况：

(1) CPU1 从一个特定的内存单元读出，并测试到其中的某一位（假定是 bit4）为 0，然后将这一位改成 1。

(2) 在 CPU1 还没有来得及把修改后的结果写回该内存单元之前，CPU2 也从同一内存单元读出，也测试到 bit4 为 0，然后也将这一位改成 1。

(3) 然后，CPU1 把修改后的结果写回内存单元。由于该内存单元的 bit4 原来是 0，CPU1 便以为取得了该项临界资源，并且确信不会有其它进程前来打扰。在 CPU1 看来，它已经把这个内存单元的 bit4 改成了 1，如果其它进程再来对此项临界资源执行“测试并设置”操作，就会因此而被关在门外。

(4) 然后，CPU2 也把修改后的结果写回内存单元，也以为取得了该项临界资源，也确信不会有其它进程可以前来打扰。

可是，实际上这两个 CPU 之间的互相干扰已是十分可能的了。这个例了告诉我们，与单处理器结构相比，SMP 结构对互斥操作的微观“分辨率”需更高，有些在单处理器结构中的“原子操作”在 SMP 结构中不再是原子的了。

第二个问题是高速缓存与内存之间（内容的）的一致性问题。在单处理结构中，使用高速缓存的目的仅在于通过提高 CPU 取指令和读／写数据的速度来改善系统的性能。高速缓存与内存的关系跟缓冲页面与磁盘上记录块的关系相似。当 CPU 要从内存读数据时，高速缓存的硬件会先根据目标地址检查这部分数据是否已经在高速缓存中，如果是，就不需要从内存中读了；否则，就从内存将目标所在的一小块数据，称为一条缓冲线（cache line），以较高的速度装入高速缓存。一旦装入了高速缓存，再要从属于同一缓冲线的内存单元读数据时就能在高速缓存中“命中”，因而不需要到内存中去读了。高速缓存中的内容也像缓冲页面那样随时间而“老化”，当高速缓存的容量不够时就把最老的缓冲线内容丢弃，从而达到周转使用高速缓存空间的目的。此外，软件也可以通过特定的指令主动将高速缓存中属于某个内存区间的内容丢弃，以后要从这个区间读时就又从内存装入。读者以前看到过，通过 DMA 操作，从外设读入以后要丢弃与 DMA 缓冲区对应的缓冲线内容，就是因为 DMA 读操作改变了内存中 DMA 缓冲区的内容，使得高速缓存与内存不一致了。DMA 操作是由设备驱动程序安排启动的，所以设备驱动程序知道应该在操作完成以后丢弃高速缓存的内容。但是，在 SMP 结构中情况就更复杂了，因为一个 CPU 并不知道别的 CPU 会在何时改变内存的内容。采用高速缓存时的写操作有两种模式，一种称为“穿透”（Write-Through）模式，在这种模式中高速缓存对于写操作就好像不存在一样，每次写时都直接写到内存中，所以实际上只是对读操作使用高速缓存，因而效率相对较低。另一种称为“回写”（Write-Back）模式，写的时候先写入高速缓存，然后由高速缓存的硬件在周转使用缓冲线时自动写入内存，或者由软件主动地“冲刷”有关的缓冲线。因此，在改变了缓冲页面的内容，并启动 DMA 写操作将其写回磁盘之前要先“冲刷”高速缓存中有关的缓冲线，因为改变了的内容可能还没有回写到内存缓冲区中。

从功能和作用的角度看，可以把高速缓存分成三部分。

(1) 对访问内存的缓冲（Cache），其作用类似于访问磁盘时的缓冲，包括对数据和指令两方面的缓冲，这一部分是本来意义上的“高速缓存”。不过，对指令的缓冲基本上是透明的，我们这里关心的只是对数据的缓冲。此外，高速缓存在物理上往往分成一级（L1）和二级（L2）两

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

部分，但是从软件的角度看并无不同。

(2) 页面映射目录和页面映射表的缓冲存储（TLB）。我们在第 2 章中曾经讲到，为了加快页面映射的速度，在页面映射的过程实际上并不是每次都到内存中访问页面映射目录和页面映射表，而是将它们所在的页面装入 CPU 内部，以加快地址转换的速度。这里所谓“ CPU 内部”其实就是高速缓存的一部分，称为 TLB，即“地址转换 / 查找缓冲区”（Translation Lookaside Buffers），因为其目的在于地址映射。

(3) 第三部分是写内存缓冲区（Write Buffer）。在多处理器系统中，当一个处理器启动对内存的写操作时，有可能系统总线已被锁住，或者其它处理器正在装入 / 写回一条缓冲线，此时 CPU 把要写的内容暂时放在缓冲区中，继续往下执行而不必停下来等待。然后，当条件允许时，缓冲区的有关硬件会把暂时缓冲着的内容写回内存。

高速缓存与内存一致性的问题，对于 TLB 和数据的缓冲在表现上和处理上有所不同。

在 CPU 中有个寄存器，称为“存储类型及范围寄存器”（Memory Type Range Register）MTRR，通过这个寄存器可以将内存中的不同区间设置成使用或不使用高速缓存，以及对于写操作采用穿透模式或回写模式。具体地，可以把每一个内存区间设置成下列几种模式：

(1) 不缓冲。
(2) 缓冲，穿透模式。
(3) 缓冲，回写模式。
(4) 写保护，只允许读不允许写。

此外，MTRR 还支持一种“结合写”（write combining）模式，用于跟写入次序无关的区间。例如，图像缓冲区就是这样的区间：把一个像素很快地先写成红，再写成黄，再写成绿，跟直接写成绿在效果上并无不同。当然，这种模式对于操作系统内核是不适合的。

其实，除 MTRR 以外，通过其它控制寄存器也能从总体上或按页面方式控制高速缓存的方式，只是没有采用 MTRR 时那样的灵活性和控制精度而已。再说在 Pentium 以前的处理器中根本就没有 MTRR，所以在 Linux 内核中是否采用 MTRR 是个选择项。

还有个与高速缓存密切相关的重要问题，就是高速缓冲的运用有可能改变对内存操作的次序。假定有两个观察者，一个观察 CPU 内部高速缓存受到访问的次序，而另一个观察内存受到访问的次序，则二者可能会有相当大的差异。前者就是程序中编排好的次序，所以称作“指令序”（program ordering）。后者则是实际出现在处理器外部，即系统总线上的次序，所以称作“处理器序”（processor ordering）。显然，不使用高速缓存时二者必然相同。而如果使用高速缓冲，那就要看具体的情况和操作。如果保证“处理器序”与“指令序”相同，就称作“强序”（strong ordering）；反之，如果“处理器序”有时候可能不同于“指令序”，就称作“弱序”（weak ordering）。对于单处理器结构的系统，这二者的不同并不成什么问题，然而对 SMP 结构的系统却可能成为问题。

此外，gcc 在编译过程中进行的优化也可能会改变操作的次序。这种情况下的“指令序”本身就可能与程序编写者的意图不同，在单处理器系统中已可能造成问题，在 SMP 结构中就更加可能了。

在 SMP 结构中，高速缓存的作用比在单处理器结构中更为重要，因为它不但可以提高取指令和读 / 写数据的速度，还有利于减少多个 CPU 在访问内存时的冲突。一般的内存都不允许在同一时间内（同一个“内存访问周期”内）受到多个 CPU 的访问，所以，在 SMP 结构中通常每个 CPU 都有较大的高速缓存，从而一旦把高速缓存装满以后，就可以运行相当长的时间而无需经常地读 / 写物理上的内存。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

如上所述，单处理器系统中的 DMA 操作都是由设备驱动程序主动地启动的，所以设备驱动程序（确切地说是程序员）知道什么时候应该丢弃哪些缓冲线的内容，什么时候应该冲刷哪些缓冲线的内容。可是在 SMP 结构中就复杂了，此时每个 CPU 都有可能改变内存中的内容，而且是异步地改变。就是说，每个 CPU 都只知道自己何时会改变内存的内容（包括自己启动的 DMA 操作），但是都不知道别的 CPU 会在什么时候改变内存的内容，也不知道自己本地的高速缓存中的内容是否已经与内存中不一致。反过来，每个 CPU 都可能因为改变了内存的内容而使其它 CPU 的高速缓存变得不一致了。

第三个问题是对中断的处理。在单处理器结构中，整个系统只有一个 CPU，所有的中断请求都由这个 CPU 响应和处理。可是，在 SMP 结构中怎么办呢？是固定让其中的某一个 CPU 处理所有的中断请求，还是让所有的 CPU 轮流处理？如果是固定让一个 CPU 处理，那么其它的 CPU 是否就没有任何中断，连称为“心跳”的时钟中断也没有了？要是那样，万一在那些 CPU 上运行的进程陷入了死循环，从而永远没有机会进行系统调用，这些 CPU 岂不是就永远不会有进程调度了？如果是让所有的 CPU 轮流处理，或者谁空闲谁处理，那又怎样把中断请求分配给不同的 CPU 呢？

还有，后面读者会看到，为了解决上述的问题，就像进程之间要有“进程间通信”的手段一样，处理器之间也要有“处理器间中断”的手段，这显然不是单靠软件就能解决的。

下面，我们将结合 Intel 的 i386 结构，主要是 Pentium 处理器，以及 Linux 内核中的有关源代码来看这些问题是如何解决的。这里要说明一点，限于本书的篇幅，我们不可能对 SMP 结构这个课题作全面而详尽的介绍，那本身就已经足够一本专著的内容。事实上，Curt Schimmel 的《UNIX Systems for Modern Architectures》一书就是这方面的经典著作。同时，我们也不可能就 Pentium 处理器对 SMP 结构的支持作详尽的介绍。需要深入研究 SMP 结构的读者应该进一步阅读有关的专著和 Intel 提供的技术资料（可以从 Intel 的网站下载）。

9.2　SMP 结构中的互斥问题

Intel 在 80386 的设计时就在一定程度上考虑到了 SMP 结构的需要。后来，随着应用中逐渐增长的需求和技术的发展，又对 i386 系统结构逐步作了加强，到推出 Pentium 处理器（更确切地说是 P6 系列）的时候就已经有了比较完整的解决方案。

先看 i386 系列的处理器，特别是 Pentium，是怎样解决 SMP 结构中对临界资源操作的原子性问题的。首先，单纯的读或写本来就是原子的。不管是 8 位、16 位，还是 32 位的读或写，都只要一个内存读 / 写微操作就能完成，所以是不可分割的。问题在于一些既要读又要写，因而需要两个或两个以上的微操作才能完成的指令，如前述的“读—改—写”操作就是一个例子。对于这样的指令，i386 CPU 提供了在指令执行期间对总线加锁的手段。CPU 芯片上有一条引线 LOCK，如果汇编语言的程序中在一条指令前面加上前缀“ LOCK”，经过汇编以后的机器代码就使 CPU 在执行这条指令的时候把引线 LOCK 的电位拉低，从而把总线锁住，这样同一总线上别的 CPU 就暂时不能通过总线访问内存了。这方面还有个特例，就是在执行指令 xchg 的时候 CPU 会自动将总线锁住，而不需要在程序中使用前缀“ LOCK”。xchg 指令将一个内存单元中的内容与一个寄存器的内容对换，因此常常用于对内核信号量

（Semephore）的操作。显然，这是一条既有读又有写的操作指令。下面的函数 spin_trylock()就是一个使用 xchg 的实例，取自 include/asm-i386/spinlock.h：

==================== include/asm-i386/spinlock.h 68 76 ====================

```
static inline int spin_trylock(spinlock_t *lock)
{
    char oldval;
    __asm__ __volatile__(
        "xchgb %b0,%1"
        :"=q" (oldval), "=m" (lock->lock)
        :"0" (0) : "memory");
    return oldval > 0;
}
```

这个函数将操作数%0，即 oldval 先设置成 0，然后与操作数%1，即内存单元 lock->lock 的内容对换。如果 lock->lock 的内容原来就是 0，就说明这个锁 lock 在此之前已经被锁上了，所以 spin_trylock()返回 0 表示加锁失败。反之，如果 lock->lock 的内容原来非 0，则现在变成了 0，加锁就成功了，所以 spin_trylock()返回 1。由于在执行 xchg 指令时 CPU 会自动锁住总线，所以不需要在这条指令前面加前缀“ LOCK”。再看对这个函数的使用（kernel/softirq.c）：

==================== kernel/softirq.c 244 251 ====================

```
spinlock_t global_bh_lock = SPIN_LOCK_UNLOCKED;

static void bh_action(unsigned long nr)
{
    int cpu = smp_processor_id();

    if (!spin_trylock(&global_bh_lock))
        goto resched;
 . . . . . .
```

==================== kernel/softirq.c 256 257 ====================

```
    if (bh_base[nr])
        bh_base[nr]();
 . . . . . .
```

==================== kernel/softirq.c 260 261 ====================

```
    spin_unlock(&global_bh_lock);
    return;
 . . . . . .
```

==================== kernel/softirq.c 265 267 ====================

```
resched:
    mark_bh(nr);
}
```

这里的 global_bh_lock 是用于 bh 函数执行机制的锁，spinlock_t 数据结构的定义在 include/asm-i386/spinlock.h 中：

==================== include/asm-i386/spinlock.h 17 26 ====================

```
/*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
18    * Your basic SMP spinlocks, allowing only a single CPU anywhere
19    */
20
21   typedef struct {
22        volatile unsigned int lock;
23   #if SPINLOCK_DEBUG
24        unsigned magic;
25   #endif
26   } spinlock_t;
```

从 250 行的 spin_trylock()至 260 行的 spin_unlock()之间是需要保证互斥、且在同一时间中只允许一个 CPU 在里面执行的临界区。所以，CPU 在进入这个区间前，先要通过 spin_trylock()加锁，而 global_bh_lock 的初始值非 0。如果 spin_trylock()的返回值为 0 就表示已经有 CPU 在里面执行，所以就跳过对 bh 函数的执行而转到 resched 处。这样，利用 xchg 指令的原子性，就实现了对临界资源 global_bh_lock 操作的互斥性，进而实现了整个临界区中操作的互斥性。

显然，在竞争（或者说抢夺）临界资源时，只有系统中所有的 CPU 都确实是在对同一个内存单元执行 xchg 指令才有意义。如果各个 CPU 只是对 global_bh_lock 在其高速缓存中的一个副本执行 xchg 指令，而执行的结果又不能立即为其它 CPU 所见，则 spin_trylock()就毫无作用了。读者在下一节中将看到，Intel 在 Pentium 处理器中提供了一种称为 snooping 的机制，能自动维持高速缓存与内存的数据一致性，使这个问题对于软件成为透明。否则，就必须把 global_bh_lock 放在一个不加缓冲的区间中。

再来看（计数）信号量的实现，函数 down()的代码读者以前看到过了，这里只列出其中几行供读者比较和领会（include/asm-i386/semaphore.h）：

==================== include/asm-i386/semaphore.h 114 115 ====================

```
114  static inline void down(struct semaphore * sem)
115  {
     . . . . . .
```

==================== include/asm-i386/semaphore.h 120 123 ====================

```
120       __asm__ __volatile__(
121            "# atomic down operation\n\t"
122            LOCK "decl %0\n\t"     /* --sem->count */
123            "js 2f\n"
     . . . . . .
```

==================== include/asm-i386/semaphore.h 129 132 ====================

```
129            :"=m" (sem->count)
130            :"c" (sem)
131            :"memory");
132  }
```

这里使 sem->count 的数值减 1 的操作是通过 dec 指令完成的。显然，这条指令执行的也是“读—改—写”操作，为了保证操作的原子性，在指令前面加上了前缀“ LOCK”。

前面讲过，高速缓存的使用可能会使实际的内存操作改变次序。从访问内存的角度看，这就使得 CPU 可能会有一些逻辑上已经完成、但是物理上尚未实现的内存操作。在 SMP 结构中，这种次序的改变也可能影响到 CPU 间的同步与互斥。所以，需要有一种手段，使得在某些操作之前把这种“欠下 ”

PDF-XCHANGE Click to buy NOW! www.docu-track.com

的内存操作全都最终地、物理地完成，就好像把欠下的债都结清，然后再开始新的（通常是比较重要的）活动一样。这种手段称为“内存路障”（memory barrier），对于 SMP 结构也是很重要的。Intel 在 Pentium 处理器中通过多种方式提供内存路障：

(1) 凡是对系统总线加锁的操作都起着内存路障的作用。所以，在 down()中的 dec 指令就是一个内存路障，CPU 在执行这条指令之前会自动把已经写入高速缓存、但是尚未最终写回内存的内容先冲刷出去。

(2) 一些特殊的指令和操作起着内存路障的作用。这样的指令有 iret、cpuid、sfence，以及以控制寄存器或程序调试寄存器为目标的 mov 指令，还有对 GDTR、LDTR、IDTR 等寄存器的装入操作和对高速缓存的控制操作。这里特别值得一提的是指令 cpuid，我们有时候会在汇编代码中看到似乎与程序逻辑无关的 cpuid 指令，那其实就是起着路障的作用。

为了编写程序的方便，内核的代码中定义了几个用作路障的宏操作，具体有 mb()、rmb()以及 wmb()，均定义于 include/asm-i386/system.h 内。对于 Intel 结构的 CPU，如 Pentium，rmb()实际上就是 mb()。

==================== include/asm-i386/system.h 257 281 ====================

```
/*
 * Force strict CPU ordering.
 * And yes, this is required on UP too when we're talking
 * to devices.
 *
 * For now, "wmb()" doesn't actually do anything, as all
 * Intel CPU's follow what Intel calls a *Processor Order*,
 * in which all writes are seen in the program order even
 * outside the CPU.
 *
 * I expect future Intel CPU's to have a weaker ordering,
 * but I'd also expect them to finally get their act together
 * and add some real memory barriers if so.
 *
 * The Pentium III does add a real memory barrier with the
 * sfence instruction, so we use that where appropriate.
 */
#ifndef CONFIG_X86_XMM
#define mb()  __asm__ __volatile__ ("lock; addl $0,0(%%esp)": : :"memory")
#else
#define mb()  __asm__ __volatile__ ("sfence": : :"memory")
#endif
#define rmb() mb()
#define wmb() __asm__ __volatile__ ("": : :"memory")

```

这里的指令 sfence 意为“存储器篱笆”（storage fence），是专为存储器同步而设的。与 cpuid 相比，sfence 没有任何副作用，而 cpuid 则会改变%eax 等寄存器的内容。但是，并不是所有的 Pentium CPU 都提供 sfence 指令，所以对不提供这条指令的 CPU 就通过一条带前缀 LOCK 的指令来实现，但是又要不产生任何副作用，所以把正在堆栈顶部的那个数据作为操作对象，在它上面加上 0。至于 wmb()，代码的作者在注释中说目前 Intel 的 CPU 对写操作自动地保证“处理器序”，所以并不需要做什么事。这

些函数还有个作用，就是使 gcc 在编译时不会试图跨过这些函数进行优化。

下面是一个在程序中调用函数 mb()设置内存路障的实例，取自 kernel/softirq.c。

==================== kernel/softirq.c 269 273 ====================

```
void init_bh(int nr, void (*routine)(void))
{
     bh_base[nr] = routine;
     mb();
}
```

这里的目的显然是要使由当前 CPU 为某个软中断设置的 bh 函数立即为系统中其它 CPU 所见，其作用与冲刷 TLB 相似。

所以，对于现代的 CPU，不同处理器对临界资源操作的互斥性和次序的正确性主要是靠硬件解决的，软件只起着辅助的作用。

顺便提一下，前面 bh_action()的代码中引用了一个 inline 函数 smp_processor_id()，目的是要知道当前进程在哪一个 CPU 上运行，与此相似的还有个 hard_smp_processor_id()，均定义于 include/asm-i386/smp.h 内：

==================== include/asm-i386/smp.h 72 84 ====================

```
/*
 * This function is needed by all SMP systems. It must _always_ be valid
 * from the initial startup. We map APIC_BASE very early in page_setup(),
 * so this is correct in the x86 case.
 */

#define smp_processor_id() (current->processor)

extern __inline int hard_smp_processor_id(void)
{
     /* we don't want to mark this access volatile - bad code generation */
     return GET_APIC_ID(*(unsigned long *)(APIC_BASE+APIC_ID));
}
```

每当一个 CPU 调度一个进程运行时，都把自己的逻辑序号设置在该进程的 task_struct 结构中的 processor 字段中。这个序号从哪儿来呢？很简单，来自调度之前原来在这个 CPU 上运行的进程。那么，第一个在此 CPU 上运行的进程（空转进程）又从哪里取得这个序号呢？对于主 CPU，这个序号是 0，而次 CPU 的序号则取决于主 CPU 启动它们运行的先后。至于 hard_smp_processor_id()，则从本地 APIC 中读出其物理序号。

在单 CPU 系统中这两个函数都固定返回 0。

9.3 高速缓存与内存的一致性

本章第一节中讲到的第二个问题是各个 CPU 私有的、局部的高速缓存与公共的、全局的内存如何同步的问题。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

对于高速缓存中的第一部分，实际上一般只有数据才有一致性的问题，因为对指令一般都是只读，而并不在运行的过程中动态地加以改变。Intel 在 Pentum CPU 中为已经装入高速缓存的数据提供了一种自动与内存保持一致的机制，称为“窥探”（snooping）。每个 CPU 内部都有一部分专门的硬件，一旦启用了高速缓存以后就时刻监视着系统总线上对内存的操作。由于对内存的操作一定要经过系统总线，所以没有一次实际访问内存的操作是能够逃过其监视的。如果发现有来自其它 CPU 的写操作，而本 CPU 的高速缓存中又缓冲存储着该次写操作的目标，就会自动把相应的缓冲线废弃，使得在需要用到这些数据时重新将其装入高速缓存，以此达到二者的一致。这样一来，SMP 结构中高速缓存与内存的数据一致性问题对于软件而言就是透明的了。显然，这种机制大大地简化了软件的设计与实现。读者可以回顾一下 SMP 结构中对临界资源加锁（见第 4 章）的过程。如果没有这种自动保持一致的机制，就必须把所有用于 spin_lock 的变量全都放在一个不加缓冲的区间，否则各个 CPU 所测试和改变的可能只是自己高速缓存中的内容，而根本起不到 spin_lock 的作用。

对于高速缓存中的 TLB 部分，问题便有所不同。这个问题可以通过 IPI、即“处理器间中断”来解决。每当一个 CPU 改变了内存中某个页面映射目录或页面映划表的内容，从而可能引起其它 CPU 的 TLB 与此不一致时，就向系统中正在使用这个映射表的 CPU 发出一个中断请求，请它们废弃各自 TLB 中的内容。后面读者将会看到，在 i386 SMP 结构中采用的是“高级可编程中断控制器 ” APIC，在 APIC 中专门为此而设立了一种中断请求，其中断向量为 INVALIDATE_TLB_VECTOR。当一个 CPU 收到这种中断请求时，就会在相应的中断服务程序中将其 TLB 中的部分或全部内容作废（Invalidate）。TLB 中的内容作废以后，就会在需要用到这些内容时自动重新装入，这样就保证了内容的一致性。同时，内核中还提供了一个通用的函数 send_IPI_mask()，当一个 CPU 需要向其它 CPU 发出某种中断请求时，就可以用相应的中断向量为参数调用这个函数来完成。对于 INVALIDATE_TLB_VECTOR 中断，内核还在 send_IPI_mask()的基础上提供了一个函数 flush_tlb_others()。当一个 CPU 要求其它 CPU 废弃各自 TLB 中的内容时，就可以调用 flush_tlb_others()达到“冲刷其它（进程的）TLB”的目的。注意这里所谓“冲刷”实际上是废弃。

我们通过一个实例来看对 flush_tlb_others()的调用。在第 2 章中，读者看到过内核线程 kswapd()通过 try_to_swap_out()试图换出某个进程的一个页面的过程。在函数 try_to_swap_out()中，先对给定表项所映射的内存页面进行了种种检查。如果决定要断开这个页面的映射，就先把这个表项清成 0，然后再根据具体的情况决定是否需要将这个表项设置成指向盘上的页面映象。下面是这个函数中的一个片断（mm/vmscan.c）。

==================== mm/vmscan.c 38 39 ====================
[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()>swap_out_pmd()>try_to_swap_out()]

```
static int try_to_swap_out(struct mm_struct * mm, struct vm_area_struct* vma, unsigned long address,
pte_t * page_table, int gfp_mask)
{
    . . . . . .
```

==================== mm/vmscan.c 78 84 ====================

```
        /* From this point on, the odds are that we're going to
         * nuke this pte, so read and clear the pte.  This hook
         * is needed on CPUs which update the accessed and dirty
         * bits in hardware.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
82         */
83        pte = ptep_get_and_clear(page_table);
84        flush_tlb_page(vma, address);
     . . . . . .
==================== mm/vmscan.c 157 157 ====================
157  }
```

这里的 83 行通过 ptep_get_and_clear()将指针 page_table 所指的表项清成 0，接着就调用 flush_tlb_page()冲刷 CPU 的高速缓存。这里要说明：kswapd()是个内核线程，正在某一个 CPU 上运行，而正在处理中的页面表却通常是属于另一个进程的，那个进程有可能正在另一个 CPU 上运行着。如果那样，则它的页面表很可能已被装入那个 CPU 的高速缓存中。所以，此时要向正在运行着目标进程的 CPU 发送一个 INVALIDATE_TLB_VECTOR 中断请求，让它废弃其高速缓存中的内容而重新装入已经改变了的页面映射表。这是由 flush_tlb_page()完成的，其代码在 arch/i386/kernel/smp.c 中：

```
==================== arch/i386/kernel/smp.c 372 386 ====================
[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()
>swap_out_pmd()>try_to_swap_out()>flush_tlb_page()]
372  void flush_tlb_page(struct vm_area_struct * vma, unsigned long va)
373  {
374       struct mm_struct *mm = vma->vm_mm;
375       unsigned long cpu_mask = mm->cpu_vm_mask & ~(1 << smp_processor_id());
376
377       if (current->active_mm == mm) {
378            if(current->mm)
379                 __flush_tlb_one(va);
380             else
381                 leave_mm(smp_processor_id());
382       }
383
384       if (cpu_mask)
385            flush_tlb_others(cpu_mask, mm, va);
386  }
```

读者应该还记得，进程的虚存空间是由一个 mm_struct 数据结构代表的。在 mm_struct 数据结构中有个字段 cpu_vm_mask，这是个位图，表示有哪一些 CPU 正在使用这个空间。系统中的每一个 CPU 都对应着这个位图中的一位，如果是单 CPU 系统则只有 bit0 有意义。当一个 CPU 在进程调度中从一个老进程切换到一个新进程的时候，就要修改这两个进程所使用的 mm_struct 数据结构中的位图，在新进程的 mm_struct 结构中把相应的标志位设置成 1，而老进程的 mm_struct 结构中则把相应的标志位设置成 0（详见 switch_mm()的代码）。每个 CPU 都可以通过一个函数 smp_processor_id()取得其在系统中的编号，这就决定了它在位图中的位置。以前，我们在第 4 章中阅读 switch_mm()的代码时有意忽略了这个细节，现在读者可以回过去看一下。上面的 375 行把正在处理的 mm_struct 结构中的位图取到变量 cpu_mask 中，但是将正在执行这个函数的 CPU 本身除外。这样，如果 cpu_mask 为非 0 就说明系统中至少还有一个 CPU 正在运行使用这个 mm_struct 结构的进程，从而正在使用相应的页面表，所以要进一步调用 flush_tlb_others()。至于代码中的 377～382 行，那只是在当前进程，就是正在当前 CPU 上运行的进程，恰好也与目标 mm_struct 结构有关的时候才执行，我们在这里对此并不关心。函数

flush_tlb_others()的代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 304 346 ====================
[kswapd()>do_try_to_free_pages()>refill_inactive()>swap_out()>swap_out_mm()>swap_out_vma()>swap_out_pgd()
>swap_out_pmd()>try_to_swap_out()>flush_tlb_page()>flush_tlb_others()]

```
static void flush_tlb_others (unsigned long cpumask, struct mm_struct *mm,
                              unsigned long va)
{
     /*
      * A couple of (to be removed) sanity checks:
      *
      * - we do not send IPIs to not-yet booted CPUs.
      * - current CPU must not be in mask
      * - mask must exist :)
      */
     if (!cpumask)
          BUG();
     if ((cpumask & cpu_online_map) != cpumask)
          BUG();
     if (cpumask & (1 << smp_processor_id()))
          BUG();
     if (!mm)
          BUG();

     /*
      * i'm not happy about this global shared spinlock in the
      * MM hot path, but we'll see how contended it is.
      * Temporarily this turns IRQs off, so that lockups are
      * detected by the NMI watchdog.
      */
     spin_lock(&tlbstate_lock);

     flush_mm = mm;
     flush_va = va;
     atomic_set_mask(cpumask, &flush_cpumask);
     /*
      * We have to send the IPI only to
      * CPUs affected.
      */
     send_IPI_mask(cpumask, INVALIDATE_TLB_VECTOR);

     while (flush_cpumask)
          /* nothing. lockup detection does not belong here */;

     flush_mm = NULL;
     flush_va = 0;
     spin_unlock(&tlbstate_lock);
}
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

内核中有个全局量 cpu_online_map，这也是个位图，记录着系统中所有的 CPU（最多 32 个），参数 cpumask 的内容只能是 cpu_online_map 的一个子集，并且不能包括当前 CPU 本身。此外，flush_mm、flush_va 以及 flush_cpumask 都是全局量，用来作为 IPI 的辅助手段。由于是全局量，系统中所有的 CPU 都能看到这些变量。同时，对这些全局量的改变必须互斥。接着就是通过 send_IPI_mask()向有关的 CPU 发送一个 INVALIDATE_TLB_VECTOR 中断请求，发送以后要等待位图 flush_cpumask 变成全 0，表示所有的目标 CPU 都已经响应了这次中断请求。我们把具体发送中断请求的过程留到下一节，这里先看看目标 CPU 在接收到这个中断请求以后干些什么。读者将会看到，与 INVALIDATE_TLB_VECTOR 相对应的中断响应程序是 smp_invalidate_interrupt()，其代码也在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 269 302 ====================

```
/*
 * TLB flush IPI:
 *
 * 1) Flush the tlb entries if the cpu uses the mm that's being flushed.
 * 2) Leave the mm if we are in the lazy tlb mode.
 */

asmlinkage void smp_invalidate_interrupt (void)
{
    unsigned long cpu = smp_processor_id();

    if (!test_bit(cpu, &flush_cpumask))
        return;
        /*
         * This was a BUG() but until someone can quote me the
         * line from the intel manual that guarantees an IPI to
         * multiple CPUs is retried _only_ on the erroring CPUs
         * its staying as a return
         *
         * BUG();
         */

    if (flush_mm == cpu_tlbstate[cpu].active_mm) {
        if (cpu_tlbstate[cpu].state == TLBSTATE_OK) {
            if (flush_va == FLUSH_ALL)
                local_flush_tlb();
            else
                __flush_tlb_one(flush_va);
        } else
            leave_mm(cpu);
    }
    ack_APIC_irq();
    clear_bit(cpu, &flush_cpumask);
}
```

内核中有个全局的 tlb_state 数据结构数组 cpu_tlbstate[]，定义于 arch/i386/kernel/smp.c：

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== arch/i386/kernel/smp.c 106 106 ====================

```
106 struct tlb_state cpu_tlbstate[NR_CPUS] = {[0 ... NR_CPUS-1] = { &init_mm, 0 }};
```

其类型定义在 **include/asm-i386/pgalloc.h** 中：

==================== include/asm-i386/pgalloc.h 214 221 ====================

```
#define TLBSTATE_OK     1
#define TLBSTATE_LAZY   2

struct tlb_state
{
    struct mm_struct *active_mm;
    int state;
};
```

数组 cpu_tlbstate[]中所有元素的初始值都是{&init_mm, 0}，106 行中的这种表示方法也是 gcc 对 C 语言的一种扩充。每个 CPU 在这个数组中都有一个 tlb_state 结构。在 switch_mm()中，每当一个 CPU 要切换到一个进程的虚存空间时，就把这个结构中的指针 active_mm 设置成指向新的 mm_struct 结构，表示这个 CPU 正在使用这个虚存空间，并且把状态 state 设置成 TLBSTATE_OK，实际上是 1。在一般的情况下，一个 CPU 的 TLB 状态总是 TLBSTATE_OK，表示如果正在使用中的页面目录或页面表内容发生了变化就要刷新 TLB 的内容。

但是，是否只要页面目录或页面表内容发生了变化就一定要刷新 TLB 的内容呢？那倒不一定。如果有把握地知道可能发生的变化绝不会影响 CPU 的运行，就没有这个必要。我们不妨这样想：内核中代码所占的页面是不会改变的，其它的页面也没有换入／换出的问题，而内核线程又没有用户空间，所以与页面的换入／换出无关。事实上，内核中可能改变页面映射的只有几种情况，一种与 vmalloc()有关。另一种与 HIGHMEM 的映射有关，还有就是与外设总线（如 PCI 总线）有关的映射。因此，只要一个内核线程与这些操作无关，那么这个内核线程就可以“任凭风浪起，稳坐钓鱼船”。所以，在一些特殊的情况下，CPU 虽然还在使用属于某个虚拟空间的页面目录或页面表，但是即使这些页面目录或页面表发生了变化也没有必要刷新 TLB 的内容。例如，在执行系统调用 exit()的过程中，即使当前进程的页面目录或页面表发生了变化，也已经没有必要更新 TLB 的内容了。还有一种情况是，当 CPU 切换到一个不具有用户空间的内核线程时，要借用在它之前运行的那个进程的 active_mm（详见“进程的调度与切换”），所以此时进程切换了，但是页面目录和页面表却没有切换。然而，在运行这个内核线程的期间，即使用户空间的页面目录或页面表发生了变化也没有必要更新 TLB 的内容，因为内核线程本来就没有用户空间。在这样的情况下，就通过一个 inlinc 函数 enter_lazy_tlb()，将当前 CPU 的 TLB 状态设置成 TLBSTATE_LAZY，表示懒得更新。这个函数的代码在 include/asm-i386/mmu_context.h 中：

==================== include/asm-i386/mmu_context.h 17 21 ====================

```
static inline void enter_lazy_tlb(struct mm_struct *mm, struct task_struct *tsk, unsigned cpu)
{
    if(cpu_tlbstate[cpu].state == TLBSTATE_OK)
        cpu_tlbstate[cpu].state = TLBSTATE_LAZY;
}
```

调用这个函数的地方主要有两处，一处是在__exit_mm()中，还有一处是在 schedule()中，其它是在

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

初始化阶段，那时候运行的也是内核线程。

对 TLB 的冲刷（其实是废弃）可以是针对整个 TLB 的，也可以是针对一个具体页面的。对整个 TLB 的冲刷由 local_flush_tlb()进行，这是个宏操作，定义于 include/asm-i386/pgalloc.h 中：

==================== include/asm-i386/pgalloc.h 199 200 ====================

```
#define local_flush_tlb() \
	__flush_tlb()
```

这里的__flush_tlb()又是个宏操作，其定义也在同一文件中：

==================== include/asm-i386/pgtable.h 37 46 ====================

```
#define __flush_tlb()						\
	do {						\
		unsigned int tmpreg;				\
							\
		__asm__ __volatile__(				\
			"movl %%cr3, %0;  # flush TLB \n"		\
			"movl %0, %%cr3;             \n"	\
			: "=r" (tmpreg)				\
			:: "memory");				\
	} while (0)
```

要废弃整个 TLB 的内容很简单，只要把指向映射目录的指针，即控制寄存器%cr3 重新装入一次就可以了。所以这里把%cr3 中的内容先读到临时变量 tmpreg 中，然后再把它写回%cr3 就成了。虽然这个操作的前后%cr3 的内容并无改变，但是对%cr3 的写操作本身就起到了废弃整个 TLB 的内容，重新装入页面目录的作用。

再看废弃某个具体页面的操作，这是由宏操作__flush_tlb_one()完成的，定义于同一文件中：

==================== include/asm-i386/pgtable.h 87 88 ====================

```
#define __flush_tlb_one(addr) \
__asm__ __volatile__("invlpg %0": :"m" (*(char *) addr))
```

指令 invlpg 将 TLB 中属于给定页面的内容废弃。TLB 是按 32 字节“缓冲线”来缓冲的，所以给定页面的内容未必全都在 TLB 中，在 TLB 中的内容也未必连续。这条指令的作用就是把 TLB 中凡是属于给定页面的内容全都废弃。

如果 CPU 当前的 TLB 状态是 TLBSTATE_LAZY，那就干脆“闭目塞听”，通过 leave_mm()退出当前 mm_struct 结构中的位图 cpu_vm_mask，以后再有类似的改变就不会向这个 CPU 发出中断请求了。这个函数的代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 219 228 ====================

[smp_invalidate_interrupt()>leave_mm()]

```
/*
 * We cannot call mmdrop() because we are in interrupt context,
 * instead update mm->cpu_vm_mask.
 */
static void inline leave_mm (unsigned long cpu)
{
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    if (cpu_tlbstate[cpu].state == TLBSTATE_OK)
        BUG();
    clear_bit(cpu, &cpu_tlbstate[cpu].active_mm->cpu_vm_mask);
}
```

所以，如果当前 CPU 的 tlb_state 数据结构中的指针指向某个 mm_struct 结构，而这个结构内的位图 cpu_vm_mask 又不包括当前 CPU，那就说明这个 CPU 的 TLB 实际上已经不一致了，只不过因为当前进程是内核线程，这种不一致不会引起什么后果而已。到下一次进程调度的时候，如果要切换到一个使用着不同页面目录的进程，就会在切换时装入新的页面目录，并且把当前 CPU 的 TLB 状态又改为 TLBSTATE_OK，这样，事情就偷懒过去了。但是，如果新的进程使用的恰好就是这个内核线程所借用的 mm_struct 结构呢？在单 CPU 结构的系统中此时不用做任何事，因为不需要切换虚存空间，TLB 中的内容又肯定是一致的。而在 SMP 结构的系统中，这时候就要检查一下了，如果发生了上述的情况就得补上一次 TLB 刷新，因为页面目录或页面表中已经发生的变化对即将运行的进程是有影响的。下面是 inline 函数 switch_mm()中的一个片段，在 include/asm-i386/mmu_context.h 中：

==================== include/asm-i386/mmu_context.h 28 30 ====================

```
static inline void switch_mm(struct mm_struct *prev, struct mm_struct *next, struct task_struct *tsk, unsigned cpu)
{
    if (prev != next) {
  . . . . . .
```

==================== include/asm-i386/mmu_context.h 45 59 ====================

```
    }
#ifdef CONFIG_SMP
    else {
        cpu_tlbstate[cpu].state = TLBSTATE_OK;
        if(cpu_tlbstate[cpu].active_mm != next)
            BUG();
        if(!test_and_set_bit(cpu, &next->cpu_vm_mask)) {
            /* We were in lazy tlb mode and leave_mm disabled
             * tlb flush IPI delivery. We must flush our tlb.
             */
            local_flush_tlb();
        }
    }
#endif
}
```

读者不妨回到第 4 章中，带着问题再读一下有关的代码。

回到 smp_invalidate_interrupt()的代码中。最后，通过 ack_APIC_irq()向 APIC 发出一个确认，然后把当前 CPU 在位图 flush_cpumask 中的对应位清 0，使发出中断请求的 CPU 能知道当前 CPU 已经废弃了 TLB 中的内容。另一方面，发出中断请求的 CPU 此时正在 flush_tlb_others()中通过一个 while 循环等待位图 flush_cpumask 变成全 0，当所有的目标 CPU 都完成了中断服务时，flush_tlb_others()的执行也就完成了。

我们在前面 flush_tlb_page()的代码中跳过了对本地 TLB 的“冲刷”，但是从代码中可以看出，具体

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

的处理同样也是由__flush_tlb()或 leave_mm()完成的。

在 try_to_swap_out()中之所以调用 flush_tlb_page()，是因为要改变了一个页面映射表的内容，而一个页面映射表正好占一个页面，所以只要废弃 TLB 中属于这个页面的内容就可以了。可是，如果改变的是页面映射目录的内容呢？虽然页面映射目录也占一个页面，但是映射目录的改变意味着整个映射的改变，因为目录中的每一项都指向一个页面映射表。所以，仅仅废弃映射目录所在的页面是不够的，此时需要废弃的是 TLB 中的全部内容。为了这个目的，内核中还有个函数 flush_tlb_mm()，用来废弃整个 TLB 的内容，其代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 358 370 ====================

```
void flush_tlb_mm (struct mm_struct * mm)
{
	unsigned long cpu_mask = mm->cpu_vm_mask & ~(1 << smp_processor_id());

	if (current->active_mm == mm) {
		if (current->mm)
			local_flush_tlb();
		else
			leave_mm(smp_processor_id());
	}
	if (cpu_mask)
		flush_tlb_others(cpu_mask, mm, FLUSH_ALL);
}
```

代码中的 local_flush_tlb()实际上就是__flush_tlb()，定义于 include/asm-i386/pgalloc.h：

==================== include/asm-i386/pgalloc.h 199 200 ====================

```
#define local_flush_tlb() \
	__flush_tlb()
```

将 flush_tlb_mm()与 flush_tlb_page()作一比较，就可以发现二者基本上是相同的，只是调用 flush_tlb_others()时的一个参数为 FLUSH_ALL，而不是具体的地址。前面我们已经看到，当参数为 FLUSH_ALL 时各个 CPU 都通过__flush_tlb()重新装入控制寄存器%cr3，这就起到了废弃 TLB 中全部内容，然后随着运行的需要重新装入的目的。

9.4 SMP 结构中的中断机制

传统的 i386 处理器都采用 8259A 中断控制器，读者在第 3 章看到过对这种器件的初始化。一般而言，8259A 的作用是提供多个外部中断源与单一 CPU 之间的连接。如果在 SMP 结构中还是采用 8259A 中断控制器，那就只能静态地把所有的外部中断源划分成若干组，分别把每一组都连接到一个 8259A，而 8259A 则与 CPU 有一对一的连接。然而这样，就达不到动态地分配中断请求的目的，也使硬件的设计变得很不简洁。因此，Intel 为 Pentiun 处理器设计了一种更为通用的中断控制器，称为“高级可编程中断控制器”（Advanced Programable Interrupt Controllor），缩写成 APIC。另一方向，考虑到“处理器间中断请求”的需要，每个 CPU 实际上都需要有个本地的 APIC，因为一个 CPU 常常要有目标地向系

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

统中的其它 CPU 发出中断请求，所以，从 Pentium 开始，Intel 就在 CPU 芯片内部集成了一个本地 APIC。但是，在 SMP 结构中还需要一个外部的、全局的 APIC，从而形成如图 9.2 所示的结构。

一般而言，集成在 CPU 芯片内部的本地 APIC 与外部的 I/O APIC 是配合使用的，实际上可以认为是将一个器件分成了两部分，既可以用于 SMP 结构，也可以用于单 CPU 结构。另一方面，在本地 APIC 中有个可以用于时钟中断源的定时器，所以即使没有 I/O APIC 的配合，也可以选择使用本地 APIC。而且，即使采用了 APIC，仍然可以把各个 CPU 单独连接到 8259A 中断控制器，图 9.2 中的“本地中断请求”就是指来自各自外部中断控制器的中断请求，以及由 CPU 内部产生的中断请求（如陷阱）。在内核的代码中，因采用本地 APIC 而需要的代码都放在条件编译选择项 CONFIG_X86_LOCAL_APIC 下面。

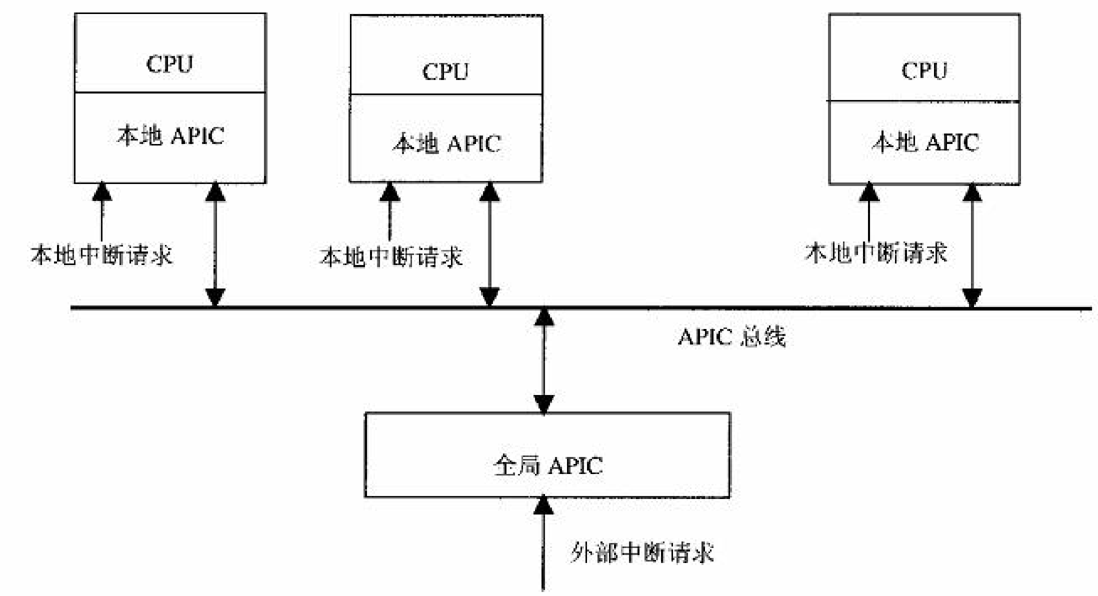


图 9.2　SMP 结构中的中断控制机构

读者在第 3 章中曾经看到，内核中的中断响应程序是通过一些宏操作利用 gcc 预处理的字符串替换和拼接功能自动生成的。同样，几个为 SMP 结构专用的中断响应程序也是由这样的宏操作生成的，这个宏操作定义于 include/asm-i386/hw_irq.h 中：

==================== include/asm-i386/hw_irq.h 123 134 ====================

```
#define BUILD_SMP_INTERRUPT(x,v) XBUILD_SMP_INTERRUPT(x,v)
#define XBUILD_SMP_INTERRUPT(x,v)\
asmlinkage void x(void); \
asmlinkage void call_##x(void); \
__asm__( \
"\n"__ALIGN_STR"\n" \
SYMBOL_NAME_STR(x) ":\n\t" \
	"pushl $"#v"\n\t" \
	SAVE_ALL \
	SYMBOL_NAME_STR(call_##x)":\n\t" \
```

```
        "call "SYMBOL_NAME_STR(smp_##x)"\n\t" \
        "jmp ret_from_intr\n");
```

文件 arch/i386/kernel/i8259.c 中引用了这个宏操作：

==================== arch/i386/kernel/i8259.c 74 83 ====================

```
74  /*
75   * The following vectors are part of the Linux architecture, there
76   * is no hardware IRQ pin equivalent for them, they are triggered
77   * through the ICC by us (IPIs)
78   */
79  #ifdef CONFIG_SMP
80  BUILD_SMP_INTERRUPT(reschedule_interrupt,RESCHEDULE_VECTOR)
81  BUILD_SMP_INTERRUPT(invalidate_interrupt,INVALIDATE_TLB_VECTOR)
82  BUILD_SMP_INTERRUPT(call_function_interrupt,CALL_FUNCTION_VECTOR)
83  #endif
```

以这里的 80 行为例，经过 gcc 的预处理以后就会变成这样：

```
asmlinkage void reschedule interrupt(void); \
asmlinkage void call_reschedule_interrupt(void); \
__asm__(
reschedule_interrupt:
       pushl $RESCHEDULE_VECTOR
       SAVE_ALL
call_smp_reschedule_interrupt:
       call smp_reschedule_interrupt
       jmp ret_from_intr
```

因此，当发生 RESCHEDULE_VECTOR 中断时，响应程序的入口是 reschedule_interrupt()，而实际处理中断响应的函数则为 smp_reschedule_interrupt()。

同样的道理，与 INVALIDATE_TLB_VECTOR 相对应的入口是 invalidate_interrupt()，而实际处理中断响应的则是 smp_invalidate_interrupt()；与 CALL_FUNCTION_VECTOR 相对应的入口是 call_function_interrupt()，而实际处理中断响应的是 smp_call_function_interrupt()。

我们在前一节中已经看过 invalidate_interrupt()的代码，这里再来看看另一个中断服务程序 smp_reschedule_interrupt()的代码。顾名思义，这个函数使 CPU 应系统中另一 CPU 之请求而进行一次进程调度，其代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 513 521 ====================

```
/*
 * Reschedule call back. Nothing to do,
 * all the work is done automatically when
 * we return from the interrupt.
 */
asmlinkage void smp_reschedule_interrupt(void)
{
     ack_APIC_irq();
}
```

从表面上看，这个函数所做的似乎只是通过 ack_APIC_irq()向 CPU 中的 APIC 发出对中断请求的确认，其代码定义于 include/asm-i386/apic.h：

==================== include/asm-i386/apic.h 54 65 ====================

```
extern inline void ack_APIC_irq(void)
{
	/*
	 * ack_APIC_irq() actually gets compiled as a single instruction:
	 * - a single rmw on Pentium/82489DX
	 * - a single write on P6+ cores (CONFIG_X86_GOOD_APIC)
	 * ... yummie.
	 */

	/* Docs say use 0 for future compatibility */
	apic_write_around(APIC_EOI, 0);
}
```

就是说，往本地 APIC 的一个寄存器中写一个 0，表示已经收到了中断请求，如此而已。可是，实际上对这种中断请求的服务隐藏在内核对中断处理的公共部分。读者在第 3 章中看到，不管是什么中断请求，内核在针对特定中断请求的服务完成以后都要检查（本 CPU）是否应该进行一次进程调度，而这正是 smp_reschedule_interrupt()所要达到的目的。当一个 CPU 需要另一个 CPU 进行一次进程调度时，就可以通过调用一个函数 smp_send_reschedule()向目标 CPU 发出一个 RESCHEDULE_VECTOR 中断请求，这个函数在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 415 418 ====================

```
void smp_send_reschedule(int cpu)
{
	send_IPI_mask(1 << cpu, RESCHEDULE_VECTOR);
}
```

中断请求的直接目的并不在于让对方进行一次进程调度，而只是使对方产生一次中断。至于目标 CPU 在执行完中断服务程序以后是否进行进程调度，那得要看事先或者处理中断的过程中是否将当前进程的 need_resched 标志设置成了 1。

相应地，在系统初始化时要在函数 init_IRQ()中为这几个中断设置好相应的中断门。我们在第 3 章中看过函数 init_IRQ()的代码，当时忽略了与 SMP 结构有关的部分，现在应该补上了。这个函数在 arch/i386/kernel/i8259.c 中：

==================== arch/i386/kernel/i8259.c 438 505 ====================

```
void __init init_IRQ(void)
{
	int i;

#ifndef CONFIG_X86_VISWS_APIC
	init_ISA_irqs();
#else
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        init_VISWS_APIC_irqs();
#endif
        /*
         * Cover the whole vector space, no vector can escape
         * us. (some of these will be overridden and become
         * 'special' SMP interrupts)
         */
        for (i = 0; i < NR_IRQS; i++) {
                int vector = FIRST_EXTERNAL_VECTOR + i;
                if (vector != SYSCALL_VECTOR)
                        set_intr_gate(vector, interrupt[i]);
        }

#ifdef CONFIG_SMP
        /*
         * IRQ0 must be given a fixed assignment and initialized,
         * because it's used before the IO-APIC is set up.
         */
        set_intr_gate(FIRST_DEVICE_VECTOR, interrupt[0]);

        /*
         * The reschedule interrupt is a CPU-to-CPU reschedule-helper
         * IPI, driven by wakeup.
         */
        set_intr_gate(RESCHEDULE_VECTOR, reschedule_interrupt);

        /* IPI for invalidation */
        set_intr_gate(INVALIDATE_TLB_VECTOR, invalidate_interrupt);

        /* IPI for generic function call */
        set_intr_gate(CALL_FUNCTION_VECTOR, call_function_interrupt);
#endif

#ifdef CONFIG_X86_LOCAL_APIC
        /* self generated IPI for local APIC timer */
        set_intr_gate(LOCAL_TIMER_VECTOR, apic_timer_interrupt);

        /* IPI vectors for APIC spurious and error interrupts */
        set_intr_gate(SPURIOUS_APIC_VECTOR, spurious_interrupt);
        set_intr_gate(ERROR_APIC_VECTOR, error_interrupt);
#endif

        /*
         * Set the clock to HZ Hz, we already have a valid
         * vector now:
         */
        outb_p(0x34,0x43);     /* binary, mode 2, LSB/MSB, ch 0 */
        outb_p(LATCH & 0xff , 0x40);    /* LSB */
        outb(LATCH >> 8 , 0x40);    /* MSB */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
494
495 #ifndef CONFIG_VISWS
496     setup_irq(2, &irq2);
497 #endif
498
499     /*
500      * External FPU? Set up irq13 if so, for
501      * original braindamaged IBM FERR coupling.
502      */
503     if (boot_cpu_data.hard_math && !cpu_has_fpu)
504         setup_irq(13, &irq13);
505 }
```

代码中 452 行的 for 循环原已设置了从 FIRST_EXTERNAL_VECTOR，即 0x20 开始的 224 个中断门向量，而如果是 SMP 系统就修改了其中的 4 个，它们分别对应着下列 4 个中断向量（见 include/asm-i386/hw_irq.h）：

==================== include/asm-i386/hw_irq.h 41 43 ====================

```
41 #define INVALIDATE_TLB_VECTOR   0xfd
42 #define RESCHEDULE_VECTOR   0xfc
43 #define CALL_FUNCTION_VECTOR    0xfb
```

==================== include/asm-i386/hw_irq.h 52 58 ====================

```
52 /*
53  * First APIC vector available to drivers: (vectors 0x30-0xee)
54  * we start at 0x31 to spread out vectors evenly between priority
55  * levels. (0x80 is the syscall vector)
56  */
57 #define FIRST_DEVICE_VECTOR 0x31
58 #define FIRST_SYSTEM_VECTOR 0xef
```

为中断向量 FIRST_DEVICE_VECTOR 设置的中断响应程序是 interrupt[0]，回顾一下第 3 章中的有关内容便可以知道，这个程序是 IRQ0x01_interrupt()。进入这个函数以后就会把数值-256，即 0xffffff00 压入堆栈，然后转入 common_interrupt，最后进入 do_IRQ()，余可类推。其中从 0x31 到 0xef 是用于外部 APIC 即 I/O APIC 的中断向量。这个区间的中断向量基本上没有什么变化，还与采用 8259A 时大致相同。然而，在 SMP 结构中这些中断请求将要由外部 APIC（而不是 8259A）送达各个 CPU，所以还要通过一个函数 setup_IO_APIC()对外部 APIC 芯片进行初始化，并关闭 8259A。与此有关的代码都在 arch/i386/kernel/io_apic.c 中。因代码过于专门，我们在这里就从略了。有兴趣或需要的读者可结合 Intel 的外部 APIC 芯片（82489DX）技术资料阅读这些代码。

此外，479～484 行为 CPU 的内部 APIC 设置中断向量，主要是时钟中断。在单 CPU 的系统中，是否采用内部 APIC 是个编译选择项。

不管是外部还是内部，APIC 都支持从 0x20 到 0xff 共 240 个不同的中断向量（0～0x1f 用于 CPU 本身的陷阱）。这些中断向量分成 15 个优先级，可以按中断向量号除于 16 算得，优先级 15 为最高。每个 CPU 都通过设置其内部 APIC 表明准备响应哪一些中断请求。不过，每个 CPU 都应将准备加以响应的中断向量尽量均匀地分布于不同的优先级中，Intel 的技术资料规定每个 CPU 在每个优先级中使用的中断向量不超过两个，否则就有可能丢失中断请求。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

外部 APIC 担负着把来自外部设备的中断请求提交和分配给系统中各个 CPU 的任务。对于每一个中断向量，可以将外部 APIC 设置成静态或动态两种不同模式之一。如果某一个中断向量是静态分配的，则外部 APIC 把这种中断请求提交给预设的一个或多个 CPU；否则就提交给优先级最低的 CPU。而各个 CPU 当前的优先级也是可以通过内部 APIC 设置的。与此有关的代码都在 arch/i386/kemel/apic.c 里，这些代码也过于专门，我们就不深入进去了。外部 APIC 的初始化是由主 CPU 通过 setup_IO_APIC() 进行的，其代码在 arch/i386/kernel/io_apic.c 中：

==================== arch/i386/kernel/io_apic.c 1521 1538 ====================

```
void __init setup_IO_APIC(void)
{
	enable_IO_APIC();

	io_apic_irqs = ~PIC_IRQS;
	printk("ENABLING IO-APIC IRQs\n");

	/*
	 * Set up the IO-APIC IRQ routing table by parsing the MP-BIOS
	 * mptable:
	 */
	setup_ioapic_ids_from_mpc();
	sync_Arb_IDs();
	setup_IO_APIC_irqs();
	init_IO_APIC_traps();
	check_timer();
	print_IO_APIC();
}
```

考虑到本书的篇幅，我们无法在这里深入到这些代码中了。事实上，光是与 APIC 有关的内容和代码就已经够写一本书了。

除外部 APIC 可以把来自外部设备的中断请求提交系统中的各个 CPU 外，每个 CPU 也都可以通过其内部 APIC 向其它 CPU 发出中断请求。当一个 CPU 要引起其它 CPU 的 INVALIDATE_TLB_VECTOR 或 RESCHEDULE_VECTOR 中断时，可以通过调用 send_IPI_mask()来达到目的。这个函数的代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 172 201 ====================

```
static inline void send_IPI_mask(int mask, int vector)
{
	unsigned long cfg;
	unsigned long flags;

	__save_flags(flags);
	__cli();

	/*
	 * Wait for idle.
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
182      */
183     apic_wait_icr_idle();
184 
185     /*
186      * prepare target chip field
187      */
188     cfg = __prepare_ICR2(mask);
189     apic_write_around(APIC_ICR2, cfg);
190 
191     /*
192      * program the ICR
193      */
194     cfg = __prepare_ICR(0, vector);
195 
196     /*
197      * Send the IPI. The write to APIC_ICR fires this off.
198      */
199     apic_write_around(APIC_ICR, cfg);
200     __restore_flags(flags);
201 }
```

CPU 的内部 APIC 有一些控制寄存器，APIC_ICR 和 APIC_ICR2 是其中的两个。要向系统中的某一个 CPU 发出中断请求时，首先要通过 apic_wait_icr_idle()，确认或等待 APIC_ICR 处于空闲状态，然后通过 __prepare_ICR() 和 __prepare_ICR2()，准备好要写入这两个寄存器的数值（arch/i386/kernel/smp.c）：

==================== arch/i386/kernel/smp.c 114 122 ====================

```
114 static inline int __prepare_ICR (unsigned int shortcut, int vector)
115 {
116     return APIC_DM_FIXED | shortcut | vector | APIC_DEST_LOGICAL;
117 }
118 
119 static inline int __prepare_ICR2 (unsigned int mask)
120 {
121     return SET_APIC_DEST_FIELD(mask);
122 }
```

寄存器 APIC_ICR2 主要用来说明发送中断请求的目标，这种目标可以是具体的 CPU，也可以是除发送者自身以外的所有 CPU，还可以是包括发送者自身在内的所有 CPU，甚至仅是发送者自身。最后，把含有发送目标的数值写入寄存器 APIC_ICR2，把含有中断向量（如 RESCHEDULE_VECTOR）的数值写入寄存器 APIC_ICR，就完成了中断请求的发送操作。

还有一个处理器间中断向量是 CALL_FUNCTION_VECTOR，用来请求目标 CPU 执行一个指定的函数。发送者先设置好一个全局的 call_data_struct 数据结构，然后向目标 CPU 发出请求。这个数据结构的定义在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 420 434 ====================

```
420 /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * Structure and data for smp_call_function(). This is designed to minimise
 * static memory requirements. It also looks cleaner.
 */
static spinlock_t call_lock = SPIN_LOCK_UNLOCKED;

struct call_data_struct {
     void (*func) (void *info);
     void *info;
     atomic_t started;
     atomic_t finished;
     int wait;
};

static struct call_data_struct * call_data;
```

数据结构中的函数指针 func 就是要求对方执行的函数，另一个指针 info 则为参数。读者在前面已经看到，对应于 CALL_FUNCTION_VECTOR 的中断服务程序是 smp_call_function_interrupt()，其代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 523 541 ====================

```
asmlinkage void smp_call_function_interrupt(void)
{
     void (*func) (void *info) = call_data->func;
     void *info = call_data->info;
     int wait = call_data->wait;

     ack_APIC_irq();
     /*
      * Notify initiating CPU that I've grabbed the data and am
      * about to execute the function
      */
     atomic_inc(&call_data->started);
     /*
      * At this point the info structure may be out of scope unless wait==1
      */
     (*func)(info);
     if (wait)
          atomic_inc(&call_data->finished);
}
```

当然，一般的函数是没有必要请其它 CPU 来执行的，系统中所有的 CPU 都可共享着同样的代码和数据。之所以要通过处理器间中断请其它 CPU 执行，是因为某个函数非得由目标 CPU 才能完成。例如，Pentium 处理器有一条指令 cpuid，通过这条指令可以查询本 CPU 是什么型号、什么版本，是否支持一些特殊的功能，当前的功能设置等等的信息。但是这条指令只能由具体的 CPU 本身执行，而不能由别的 CPU 代替。这样，如果要知道系统中某一个 CPU 的有关情况，就只能通过这种处理器间中断来实现。具体的函数为 cpuid_smp_cpuid()，其代码在 arch/i386/kemel/cupid.c 中，读者可以自己阅读。

有时候需要把中断请求发送给除当前 CPU 自身以外的所有 CPU，此时可以通过

send_IPI_allbutself()，发出广播式的中断请求。其代码在 arch/i386/kernel/smp.c 中：

==================== arch/i386/kernel/smp.c 151 160 ====================

```
static inline void send_IPI_allbutself(int vector)
{
	/*
	 * if there are no other CPUs in the system then
	 * we get an APIC send error if we try to broadcast.
	 * thus we have to avoid sending IPIs in this case.
	 */
	if (smp_num_cpus > 1)
		__send_IPI_shortcut(APIC_DEST_ALLBUT, vector);
}
```

其体的发送操作由__send_IPI_shortcut()完成，其代码也在同一文件中：

==================== arch/i386/kernel/smp.c 124 149 ====================

[send_IPI_allbutself()>__send_IPI_shortcut()]

```
static inline void __send_IPI_shortcut(unsigned int shortcut, int vector)
{
	/*
	 * Subtle. In the case of the 'never do double writes' workaround
	 * we have to lock out interrupts to be safe.  As we don't care
	 * of the value read we use an atomic rmw access to avoid costly
	 * cli/sti.  Otherwise we use an even cheaper single atomic write
	 * to the APIC.
	 */
	unsigned int cfg;

	/*
	 * Wait for idle.
	 */
	apic_wait_icr_idle();

	/*
	 * No need to touch the target chip field
	 */
	cfg = __prepare_ICR(shortcut, vector);

	/*
	 * Send the IPI. The write to APIC_ICR fires this off.
	 */
	apic_write_around(APIC_ICR, cfg);
}
```

可见，广播式的中断请求发送操作要略为简单一些。

在所有的中断请求中，时钟中断扮演着特殊的重要角色。以前讲过，人们把时钟中断比喻作“心跳”。在 SMP 结构的系统中，内部 APIC 本身就包含了一个 32 位可编程定时器，所以各个 CPU 可以选择使用本地的时钟中断源，也可以选择使用外部的全局时钟中断源。所谓可编程定时器实质上是个计

数器，设置好计数器的初值以后，每来一个时钟脉冲计数器就减 1，减到 0 时就发出一个中断请求。在系统初始化阶段，主 CPU 在启动次 CPU 运行以后便通过 setup_APIC_clocks()设置 APIC 中的时钟中断源，这个函数的代码在 arch/i386/kernel/apic.c 中：

==================== arch/i386/kernel/apic.c 597 611 ====================
[start_kernel()>smp_init()>smp_boot_cpus()>setup_APIC_clocks()]

```
void __init setup_APIC_clocks (void)
{
	__cli();

	calibration_result = calibrate_APIC_clock();
	/*
	 * Now set up the timer for real.
	 */
	setup_APIC_timer((void *)calibration_result);

	__sti();

	/* and update all other cpus */
	smp_call_function(setup_APIC_timer, (void *)calibration_result, 1, 1);
}
```

在此之前，系统先设置了一个外部时钟中断源供所有 CPU 共享，并且以此为基准测算各个 CPU 的运算速度。到初始化基本完成时再通过这个函数为各个 CPU 设置其内部 APIC 中的定时器，以作为本地的时钟中断源。这里，先通过 calibrate_APIC_clock()，测算出时钟中断相对于 APIC 总线上时钟脉冲的周期。所有 CPU 的 APIC 都通过 APIC 总线连在一起，它们都有相同的时钟脉冲周期，根据这个时钟脉冲周期就可以计算出时钟中断的周期。具体的设置是由 setup_APIC_timer()完成的。当然，各个 CPU 只能设置其自己的 APIC，而不能直接设置其它 CPU 的 APIC，所以通过 smp_call_function()向所有的次 CPU 都发出一个处理器间中断请求，让各个次 CPU 也来执行这个函数。其函数代码也在 arch/i386/kernel/apic.c 中：

==================== arch/i386/kernel/apic.c 469 516 ====================

```
void setup_APIC_timer(void * data)
{
	unsigned int clocks = (unsigned int) data, slice, t0, t1;
	unsigned long flags;
	int delta;

	__save_flags(flags);
	__sti();
	/*
	 * ok, Intel has some smart code in their APIC that knows
	 * if a CPU was in 'hlt' lowpower mode, and this increases
	 * its APIC arbitration priority. To avoid the external timer
	 * IRQ APIC event being in synchron with the APIC clock we
	 * introduce an interrupt skew to spread out timer events.
	 *
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	 * The number of slices within a 'big' timeslice is smp_num_cpus+1
	 */

	slice = clocks / (smp_num_cpus+1);
	printk("cpu: %d, clocks: %d, slice: %d\n",
		smp_processor_id(), clocks, slice);

	/*
	 * Wait for IRQ0's slice:
	 */
	wait_8254_wraparound();

	__setup_APIC_LVTT(clocks);

	t0 = apic_read(APIC_TMICT)*APIC_DIVISOR;
	/* Wait till TMCCT gets reloaded from TMICT... */
	do {
		t1 = apic_read(APIC_TMCCT)*APIC_DIVISOR;
		delta = (int)(t0 - t1 - slice*(smp_processor_id()+1));
	} while (delta >= 0);
	/* Now wait for our slice for real. */
	do {
		t1 = apic_read(APIC_TMCCT)*APIC_DIVISOR;
		delta = (int)(t0 - t1 - slice*(smp_processor_id()+1));
	} while (delta < 0);

	__setup_APIC_LVTT(clocks);

	printk("CPU%d<T0:%d,T1:%d,D:%d,S:%d,C:%d>\n",
			smp_processor_id(), t0, t1, delta, slice, clocks);

	__restore_flags(flags);
}
```

为了不让所有的 CPU 都在同一时刻发生时钟中断，这里通过两个 do-while 循环根据逻辑 CPU 号引入不同数量的延迟，使各个 CPU 的时钟中断在相位上互相错开，使这些中断均匀地分布在时钟中断的周期中。

我们把各个 CPU 对本地时钟中断的服务程序列出于下（arch/i386/kernel/apic.c），供读者结合第 3 章中与时钟中断有关的内容自行阅读。

==================== arch/i386/kernel/apic.c 709 731 ====================

```
void smp_apic_timer_interrupt(struct pt_regs * regs)
{
	int cpu = smp_processor_id();

	/*
	 * the NMI deadlock-detector uses this.
	 */
```

```
	apic_timer_irqs[cpu]++;

	/*
	 * NOTE! We'd better ACK the irq immediately,
	 * because timer handling can be slow.
	 */
	ack_APIC_irq();
	/*
	 * update_process_times() expects us to have done irq_enter().
	 * Besides, if we don't timer interrupts ignore the global
	 * interrupt lock, which is the WrongThing (tm) to do.
	 */
	irq_enter(cpu, 0);
	smp_local_timer_interrupt(regs);
	irq_exit(cpu, 0);
}
```

函数 smp_local_timer_interrupt()的代码也在 arch/i386/kernel/apic.c 中：

==================== arch/i386/kernel/apic.c 643 697 ====================

```
/*
 * Local timer interrupt handler. It does both profiling and
 * process statistics/rescheduling.
 *
 * We do profiling in every local tick, statistics/rescheduling
 * happen only every 'profiling multiplier' ticks. The default
 * multiplier is 1 and it can be changed by writing the new multiplier
 * value into /proc/profile.
 */

inline void smp_local_timer_interrupt(struct pt_regs * regs)
{
	int user = user_mode(regs);
	int cpu = smp_processor_id();

	/*
	 * The profiling function is SMP safe. (nothing can mess
	 * around with "current", and the profiling counters are
	 * updated with atomic operations). This is especially
	 * useful with a profiling multiplier != 1
	 */
	if (!user)
		x86_do_profile(regs->eip);

	if (--prof_counter[cpu] <= 0) {
		/*
		 * The multiplier may have changed since the last time we got
		 * to this point as a result of the user writing to
		 * /proc/profile. In this case we need to adjust the APIC
```

```
	  * timer accordingly.
	  *
	  * Interrupts are already masked off at this point.
	  */
	 prof_counter[cpu] = prof_multiplier[cpu];
	 if (prof_counter[cpu] != prof_old_multiplier[cpu]) {
		 __setup_APIC_LVTT(calibration_result/prof_counter[cpu]);
		 prof_old_multiplier[cpu] = prof_counter[cpu];
	 }

#ifdef CONFIG_SMP
	 update_process_times(user);
#endif
	}

	/*
	 * We take the 'long' return path, and there every subsystem
	 * grabs the aproprite locks (kernel lock/ irq lock).
	 *
	 * we might want to decouple profiling from the 'long path',
	 * and do the profiling totally in assembly.
	 *
	 * Currently this isn't too much of an issue (performance wise),
	 * we can take more than 100K local irqs per second on a 100 MHz P5.
	 */
}
```

此外，读者在第 3 章中阅读过一些与时钟中断有关的代码，其中有些条件编译也是与 SMP 结构或内部 APIC 有关的，不妨回过去看一下。

9.5 SMP 结构中的进程调度

在单 CPU 的系统中，每一个给定的时刻只有当前进程是在运行中，其它所有的进程都不在运行。可是，在 SMP 结构的系统中却同时有好几个进程在运行，所以需要在进程的 task_struct 数据结构中加上两个字段。一个是 has_cpu，为 1 时表示进程正在某个 CPU 上运行，为 0 时则表示进程不在运行。另一个字段是 processor，当 has_cpu 为 1 时这个字段说明进程在哪一个 CPU 上运行。可想而知，一个进程只有在 has_cpu 字段为 0 才可以接受调度进入运行，这一点从宏操作 can_schedule()的定义可以看出（kernel/sched.c）：

==================== kernel/sched.c 108 119 ====================

```
#ifdef CONFIG_SMP

#define idle_task(cpu) (init_tasks[cpu_number_map(cpu)])
#define can_schedule(p,cpu) ((!(p)->has_cpu) && \
			((p)->cpus_allowed & (1 << cpu)))
```

```
113
114  #else
115
116  #define idle_task(cpu) (&init_task)
117  #define can_schedule(p,cpu) (1)
118
119  #endif
```

这里的 cpus_allowed 是 task_struct 数据结构中的另一个字段，它是一个位图。位图中的某一位为 1 就表示允许这个进程接受调度在相应的 CPU 上运行。从这个意义上讲，进程调度是个双向选择的过程。

当一个 CPU 通过 schedule()从系统的就绪队列中挑选了一个进程作为运行的下一个进程 next，即将从当前进程 prev 切换到这个进程时，就将其 task_struct 结构中的 has_cpu 字段设置成 1，并将 processor 设置成该 CPU 的逻辑编号。我们回过头去（第 4 章）看一下 schedule()中有关的片段（kernel/sched.c）：

==================== kernel/sched.c 508 509 ====================

```
508  asmlinkage void schedule(void)
509  {
     . . . . . .
```

==================== kernel/sched.c 518 518 ====================

```
518      this_cpu = prev->processor;
     . . . . . .
```

==================== kernel/sched.c 587 595 ====================

```
587  #ifdef CONFIG_SMP
588      next->has_cpu = 1;
589      next->processor = this_cpu;
590  #endif
591      spin_unlock_irq(&runqueue_lock);
592
593      if (prev == next)
594          goto same_process;
595
     . . . . . .
```

==================== kernel/sched.c 648 649 ====================

```
648      switch_to(prev, next, prev);
649      __schedule_tail(prev);
     . . . . . .
```

==================== kernel/sched.c 690 690 ====================

```
690  }
```

从进程 prev 切换到 next 以后，要对 prev 调用一个函数__schedule_tail()。这个函数的代码对于 SMP 结构和单 CPU 结构有较大的不同（见 kernel/sched.c）。

==================== kernel/sched.c 426 491 ====================

[schedule()>__schedule_tail()]

```
426  static inline void __schedule_tail(struct task_struct *prev)
427  {
428  #ifdef CONFIG_SMP
429      int policy;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    /*
     * prev->policy can be written from here only before `prev'
     * can be scheduled (before setting prev->has_cpu to zero).
     * Of course it must also be read before allowing prev
     * to be rescheduled, but since the write depends on the read
     * to complete, wmb() is enough. (the spin_lock() acquired
     * before setting has_cpu is not enough because the spin_lock()
     * common code semantics allows code outside the critical section
     * to enter inside the critical section)
     */
    policy = prev->policy;
    prev->policy = policy & ~SCHED_YIELD;
    wmb();

    /*
     * fast path falls through. We have to clear has_cpu before
     * checking prev->state to avoid a wakeup race - thus we
     * also have to protect against the task exiting early.
     */
    task_lock(prev);
    prev->has_cpu = 0;
    mb();
    if (prev->state == TASK_RUNNING)
        goto needs_resched;

out_unlock:
    task_unlock(prev); /* Synchronise here with release_task() if prev is TASK_ZOMBIE */
    return;

    /*
     * Slow path - we 'push' the previous process and
     * reschedule_idle() will attempt to find a new
     * processor for it. (but it might preempt the
     * current process as well.) We must take the runqueue
     * lock and re-check prev->state to be correct. It might
     * still happen that this process has a preemption
     * 'in progress' already - but this is not a problem and
     * might happen in other circumstances as well.
     */
needs_resched:
    {
        unsigned long flags;

        /*
         * Avoid taking the runqueue lock in cases where
         * no preemption-check is necessery:
         */
        if ((prev == idle_task(smp_processor_id())) ||
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
479                             (policy & SCHED_YIELD))
480                     goto out_unlock;
481
482             spin_lock_irqsave(&runqueue_lock, flags);
483             if (prev->state == TASK_RUNNING)
484                 reschedule_idle(prev);
485             spin_unlock_irqrestore(&runqueue_lock, flags);
486             goto out_unlock;
487         }
488 #else
489         prev->policy &= ~SCHED_YIELD;
490 #endif /* CONFIG_SMP */
491 }
```

从代码中可见，这个 inline 函数对于单 CPU 系统只有一行，那就是 489 行，将原来的当前进程 prev 的 SCHED_YIELD 标志位设成 0（礼让只是一次有效），而这只在 prev 通过系统调用自愿礼让，暂时放弃运行时才有实际的作用。可是，对于 SMP 结构就不同了，除了也要把进程 prev 的 SCHED_YIELD 标志位设成 0 以外，这里还有几件事要做。首先是要把进程 prev 的 has_cpu 标志设成 0，表示这个进程已不在任何 CPU 上运行。代码中的 443 行和 452 行分别在写操作以后和读操作之前设置了内存路障，目的是：一方面要使其它 CPU 立即就能看到所作的改变，另一方面是要读到内存中最新的内容。另一件事是：如果进程 prev 的运行是被剥夺的，那就替它找找出路，看是否能让它继续在另一个 CPU 上运行。进程 prev 的状态为 TASK_RUNNING，说明它的运行是被剥夺的，或者是通过系统调用自愿礼让而暂时放弃的，所以，如果在 479 行排除了礼让的可能性，就必然是被剥夺了运行。不过，被剥夺运行的进程并不一定有必要，或者不一定可以为它另找一个处理器。如果这个进程是“空转”进程，也就是当前 CPU 上的 init 进程（参阅 SMP 结构的引导过程一节）的话，那一方面本来就无事可做，另一方面也根本不允许转到其它 CPU 上运行。把这些情况都排除了以后，就只剩下了一种可能，那就是这个进程确是有事可干，只是因为强制性调度被剥夺了运行。所以，此时要调用 reschedule_idle()，尝试能否将其转到其它 CPU 上运行。这个函数的代码在 kernel/sched.c 中：

==================== kernel/sched.c 205 294 ====================

[schedule()>__schedule_tail()>reschedule_idle()]

```
static void reschedule_idle(struct task_struct * p)
{
#ifdef CONFIG_SMP
        int this_cpu = smp_processor_id();
        struct task_struct *tsk, *target_tsk;
        int cpu, best_cpu, i, max_prio;
        cycles_t oldest_idle;

        /*
         * shortcut if the woken up task's last CPU is
         * idle now.
         */
        best_cpu = p->processor;
        if (can_schedule(p, best_cpu)) {
```

```
        tsk = idle_task(best_cpu);
        if (cpu_curr(best_cpu) == tsk) {
            int need_resched;
send_now_idle:
            /*
             * If need_resched == -1 then we can skip sending
             * the IPI altogether, tsk->need_resched is
             * actively watched by the idle thread.
             */
            need_resched = tsk->need_resched;
            tsk->need_resched = 1;
            if ((best_cpu != this_cpu) && !need_resched)
                smp_send_reschedule(best_cpu);
            return;
        }
    }

    /*
     * We know that the preferred CPU has a cache-affine current
     * process, lets try to find a new idle CPU for the woken-up
     * process. Select the least recently active idle CPU. (that
     * one will have the least active cache context.) Also find
     * the executing process which has the least priority.
     */
    oldest_idle = (cycles_t) -1;
    target_tsk = NULL;
    max_prio = 1;

    for (i = 0; i < smp_num_cpus; i++) {
        cpu = cpu_logical_map(i);
        if (!can_schedule(p, cpu))
            continue;
        tsk = cpu_curr(cpu);
        /*
         * We use the first available idle CPU. This creates
         * a priority list between idle CPUs, but this is not
         * a problem.
         */
        if (tsk == idle_task(cpu)) {
            if (last_schedule(cpu) < oldest_idle) {
                oldest_idle = last_schedule(cpu);
                target_tsk = tsk;
            }
        } else {
            if (oldest_idle == -1ULL) {
                int prio = preemption_goodness(tsk, p, cpu);

                if (prio > max_prio) {
                    max_prio = prio;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
268                         target_tsk = tsk;
269                     }
270                 }
271             }
272         }
273         tsk = target_tsk;
274         if (tsk) {
275             if (oldest_idle != -1ULL) {
276                 best_cpu = tsk->processor;
277                 goto send_now_idle;
278             }
279             tsk->need_resched = 1;
280             if (tsk->processor != this_cpu)
281                 smp_send_reschedule(tsk->processor);
282         }
283         return;
284 
285 
286     #else /* UP */
287         int this_cpu = smp_processor_id();
288         struct task_struct *tsk;
289 
290         tsk = cpu_curr(this_cpu);
291         if (preemption_goodness(tsk, p, this_cpu) > 1)
292             tsk->need_resched = 1;
293     #endif
294     }
```

代码中的 208～283 行用于 SMP 结构。这段代码先检查被剥夺运行的进程是否可以在原来的 CPU 上恢复运行（如果这个 CPU 上的当前进程是“空转”进程的话）。不行就进一步通过一个 for 循环依次考察系统中的所有 CPU，如果被剥夺运行的进程允许在某个 CPU 上运行，而这个 CPU 上的当前进程是“空转”进程，或者其“ goodness”，即运行资格低于被剥夺运行的进程，那么这个 CPU 上的当前进程就是一个可以剥夺的候选对象。当系统中存在多个可以剥夺的进程时，就从中找出运行时间最长或运行资格最低的进程，总之是“大鱼吃小鱼”，“柿子拣软的捏”。到 for 循环结束时，如果找到了可以剥夺的进程，就将其 task_struct 结构中的 need_resched 标志设成 1，再向其所在的 CPU 发出一个 RESCHEDULE_VECTOR 中断请求；否则就只好算了。

信号的发送也与进程调度有关。在将一个信号发送给一个进程以后，要通过 signal_wake_up()把目标进程唤醒。在单 CPU 的系统中，如果目标进程正在运行，那就必定是同一 CPU 上的当前进程，当 CPU 从中断或系统调用返回用户空间的前夕，就会处理这个信号。否则，目标进程也许是就绪进程；也许是个正在睡眠的进程，那就要将其唤醒。总之，到目标进程下一次受调度运行时就会先处理接收到的信号。在 SMP 结构中的情况要略为复杂一些，因为目标进程有可能正在另一个 CPU 上运行。为了使信号及时得到处理，此时要向正在执行目标进程的 CPU 发送一个 RESCHEDULE_VECTOR 中断请求。下面是有关的代码（kernel/signal.c）：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== kernel/signal.c 466 491 ====================

```
466  static inline void signal_wake_up(struct task_struct *t)
467  {
468      t->sigpending = 1;
469
470      if (t->state & TASK_INTERRUPTIBLE) {
471          wake_up_process(t);
472          return;
473      }
474
475  #ifdef CONFIG_SMP
476      /*
477       * If the task is running on a different CPU
478       * force a reschedule on the other CPU to make
479       * it notice the new signal quickly.
480       *
481       * The code below is a tad loose and might occasionally
482       * kick the wrong CPU if we catch the process in the
483       * process of changing - but no harm is done by that
484       * other than doing an extra (lightweight) IPI interrupt.
485       */
486      spin_lock(&runqueue_lock);
487      if (t->has_cpu && t->processor != smp_processor_id())
488          smp_send_reschedule(t->processor);
489      spin_unlock(&runqueue_lock);
490  #endif /* CONFIG_SMP */
491  }
```

这里的第 488 行向对方发出一个 RESCHEDULE_VECTOR 中断请求。中断请求的直接目的并不在于让对方进行一次进程调度，而只是使对方产生一次中断，为及时处理刚投递的信号创造条件。

9.6 SMP 系统的引导

SMP 结构中的所有 CPU 都是平等的，没有主次之分，这实际上是建立在系统中有多个进程或者多个执行“上下文”前提下的。在同一时间中，一个“上下文”只能由一个 CPU 处理，否则只会把事情搞糟。如果系统中一共才只有一个“上下文”，那么有再多的 CPU 存在也无从发挥作用。所以，系统的引导和初始化阶段是个特例，因为在这个阶段里系统中只有一个“上下文”，只能由一个处理器来处理。在这个阶段里，也就是在系统刚加电或“总清”（reset）之后，系统中暂时只有一个处理器运行，这个处理器称为“引导处理器 ” BP；其余的处理器则处于暂停状态，称为“应用处理器 ” AP。“引导处理器”完成了系统的引导和初始化，并创建起多个进程，从而可以由多个处理器同时参与处理时，才启动所有的“应用处理器”，让它们在完成自身的初始化以后投入运行。一旦各个“应用处理器”都已投入运行，这种暂时的主次关系便告结束，从此以后便一律平等了。系统的引导和初始化本是下一章的题材，但是我们在这里关心的是“引导处理器”怎样为各个“应用处理器”作好运行的准备，然后启动其运行的过程。这个过程固然是整个系统初始化过程中的一部分，但是实际上与 SMP 结构的关

PDF-XCHANGE Click to buy NOW! www.docu-track.com

系更为密切，所以放在本章中叙述。

在初始化阶段，“引导处理器”先完成自身的初始化，进入保护模式并开启页式存储管理机制，再完成系统特别是内存的初始化，然后就从 start_kernel()调用 smp_init()进行 SMP 结构的初始化。这个函数的代码在 init/main.c 中：

==================== init/main.c 505 512 ====================

[start_kernel()>smp_init()]

```
/* Called by boot processor to activate the rest. */
static void __init smp_init(void)
{
     /* Get other processors into their bootup holding patterns. */
     smp_boot_cpus();
     smp_threads_ready=1;
     smp_commence();
}
```

这个函数的主体是 smp_boot_cpus()，它依次启动系统中的各个 CPU，让它们各自走过初始化的第一阶段。各个次 CPU 在完成了自身的初始化以后都要停下来等待一个统一的“起跑”命令。而主 CPU，则在完成了所有次 CPU 的启动以后通过 smp_commence()发出这个命令。我们先看 smp_boot_cpus()，其代码在 arch/i386/kernel/smpboot.c 中。我们分段阅读。

==================== arch/i386/kernel/smpboot.c 829 880 ====================

[start_kernel()>smp_init()>smp_boot_cpus()]

```
void __init smp_boot_cpus(void)
{
     int apicid, cpu;

#ifdef CONFIG_MTRR
     /*  Must be done before other processors booted  */
     mtrr_init_boot_cpu ();
#endif
     /*
      * Initialize the logical to physical CPU number mapping
      * and the per-CPU profiling counter/multiplier
      */

     for (apicid = 0; apicid < NR_CPUS; apicid++) {
          x86_apicid_to_cpu[apicid] = -1;
          prof_counter[apicid] = 1;
          prof_old_multiplier[apicid] = 1;
          prof_multiplier[apicid] = 1;
     }

     /*
      * Setup boot CPU information
      */
     smp_store_cpu_info(0); /* Final full version of the data */
     printk("CPU%d: ", 0);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	print_cpu_info(&cpu_data[0]);

	/*
	 * We have the boot CPU online for sure.
	 */
	set_bit(0, &cpu_online_map);
	x86_apicid_to_cpu[boot_cpu_id] = 0;
	x86_cpu_to_apicid[0] = boot_cpu_id;
	global_irq_holder = 0;
	current->processor = 0;
	init_idle();
	smp_tune_scheduling();

	/*
	 * If we couldnt find an SMP configuration at boot time,
	 * get out of here now!
	 */
	if (!smp_found_config) {
		printk(KERN_NOTICE "SMP motherboard not detected. Using dummy APIC emulation.\n");
#ifndef CONFIG_VISWS
		io_apic_irqs = 0;
#endif
		cpu_online_map = phys_cpu_present_map = 1;
		smp_num_cpus = 1;
		goto smp_done;
	}

```

先是一些准备工作。前面提到过，对内存的高速缓冲可以通过一个“存储类型及范围寄存器”，即 MTRR 加以分区管理。例如某一个区间采用“穿透”写模式，而另一个区间采用“回写”模式，再另一个区间则根本就不缓冲，等等。MTRR 并不是非用不可的，它的使用只是使管理更加精细而已，所以是否采用 MTRR 是个编译选择项。但是，如果选择了使用 MTRR，就要在启动次 CPU 之前先通过 mtrr_init_boot_cpu()完成对主 CPU 的 MTRR 的初始化。此外，内核中还有一些以 CPU 编号为下标的数组，其中 x86_apicid_to_cpu[]用于逻辑 CPU 号到物理 CPU 号的转换，还有些是为统计信息而设的，对这些数组都要加以初始化。然后，还要调用 smp_store_cpu_info()，从 CPU 收集很多信息，并根据收集的信息进行一些必要的操作。这些信息大多是通过指令 cpuid 收集的，通过这条指令可以收集到许多有关 CPU 本身的信息。例如，这些信息中不但包括 CPU 的型号，还包括由哪一家厂商制造。如果发现 CPU 是由 AMD 制造的，就要相应地调用一个针对 AMD 处理器特点的初始化函数。这些信息存贮在一个 cpuinfo_x86 结构数组 cpu_data[]中，这个数组中的内容可以通过特殊文件系统/proc 读取。在/proc 目录下有个文件/proc/cpuinfo，对这个特殊文件的读操作就是从 cpu_data[]中读出的。读者不妨试一下“ more /proc/cpuinfo”，看看你的机器用的是什么 CPU。

主 CPU 的逻辑号总是 0，而物理号则在一个全局量 boot_cpu_id 中，数组 x86_apicid_to_cpu[]和 x86_cpu_to_apicid[]提供了二者间的转换。内核中还有个全局量 smp_found_config，由主 CPU 在一个函数 setup_arch()中调用 find_smp_config()，根据 BIOS 提供的信息设置，为 0 表示系统中只有一个 CPU，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

否则就是多 CPU 系统。所以，如果此时 smp_found_config 为 0，就结束了 smp_boot_cpus()的执行；否则就继续往下执行。

==================== arch/i386/kernel/smpboot.c 881 928 ====================
[start_kernel()>smp_init()>smp_boot_cpus()]

```
	/*
	 * Should not be necessary because the MP table should list the boot
	 * CPU too, but we do it for the sake of robustness anyway.
	 */
	if (!test_bit(boot_cpu_id, &phys_cpu_present_map)) {
		printk("weird, boot CPU (#%d) not listed by the BIOS.\n",
								boot_cpu_id);
		phys_cpu_present_map |= (1 << hard_smp_processor_id());
	}

	/*
	 * If we couldn't find a local APIC, then get out of here now!
	 */
	if (APIC_INTEGRATED(apic_version[boot_cpu_id]) &&
	    !test_bit(X86_FEATURE_APIC, boot_cpu_data.x86_capability)) {
		printk(KERN_ERR "BIOS bug, local APIC #%d not detected!...\n",
			boot_cpu_id);
		printk(KERN_ERR "... forcing use of dummy APIC emulation. (tell your hw vendor)\n");
#ifndef CONFIG_VISWS
		io_apic_irqs = 0;
#endif
		cpu_online_map = phys_cpu_present_map = 1;
		smp_num_cpus = 1;
		goto smp_done;
	}

	verify_local_APIC();

	/*
	 * If SMP should be disabled, then really disable it!
	 */
	if (!max_cpus) {
		smp_found_config = 0;
		printk(KERN_INFO "SMP mode deactivated, forcing use of dummy APIC emulation.\n");
#ifndef CONFIG_VISWS
		io_apic_irqs = 0;
#endif
		cpu_online_map = phys_cpu_present_map = 1;
		smp_num_cpus = 1;
		goto smp_done;
	}

	connect_bsp_APIC();
	setup_local_APIC();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

    if (GET_APIC_ID(apic_read(APIC_ID)) != boot_cpu_id)
        BUG();

```

这里还有一系列检验。其中第 894 行检查主 CPU 是否带有内部 APIC。对于 SMP 结构，CPU 带有内部 APIC 是个必要条件。随后，又对主 CPU 的内部 APIC 进行了初始化。限于篇幅，我们不能对调用的子程序一一加以说明了，有需要的读者可以结合 Intel 的技术资料自己阅读。这里的 phys_cpu_present_map 是个全局的 CPU 位图，由主 CPU 在 setup_arch()（见第 10 章）中通过 get_smp_config()辗转调用 MP_processor_info()，根据 BIOS 提供的信息设置。

至此，所有的准备工作都已完成，下面就要逐个地启动系统中的次 CPU 了。

==================== arch/i386/kernel/smpboot.c 929 955 ====================
[start_kernel()>smp_init()>smp_boot_cpus()]

```
    /*
     * Now scan the CPU present map and fire up the other CPUs.
     */
    Dprintk("CPU present map: %lx\n", phys_cpu_present_map);

    for (apicid = 0; apicid < NR_CPUS; apicid++) {
        /*
         * Don't even attempt to start the boot CPU!
         */
        if (apicid == boot_cpu_id)
            continue;

        if (!(phys_cpu_present_map & (1 << apicid)))
            continue;
        if ((max_cpus >= 0) && (max_cpus <= cpucount+1))
            continue;

        do_boot_cpu(apicid);

        /*
         * Make sure we unmap all failed CPUs
         */
        if ((x86_apicid_to_cpu[apicid] == -1) &&
                (phys_cpu_present_map & (1 << apicid)))
            printk("phys CPU #%d not responding - cannot use it.\n",apicid);
    }

```

内核中有个全局量 max_cpus，表示系统中有多少个 CPU 其值就是多少，但是也可以在系统引导命令行中指定只用其中几个。此外，这里的 cpucount 是个计数器，初值为 0。代码中的 for 循环根据位图 phys_cpu_present_map 依次对各个“应用处理器”调用 do_boot_cpu()，为其投入运行作好准备，并启动其运行。这个函数的代码在 arch/i386/kernel/smpboot.c 中。由于比较长，我们也只好分段阅读。

==================== arch/i386/kernel/smpboot.c 541 580 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()]

```
static void __init do_boot_cpu (int apicid)
{
	struct task_struct *idle;
	unsigned long send_status, accept_status, boot_status, maxlvt;
	int timeout, num_starts, j, cpu;
	unsigned long start_eip;

	cpu = ++cpucount;
	/*
	 * We can't use kernel_thread since we must avoid to
	 * reschedule the child.
	 */
	if (fork_by_hand() < 0)
		panic("failed fork for CPU %d", cpu);

	/*
	 * We remove it from the pidhash and the runqueue
	 * once we got the process:
	 */
	idle = init_task.prev_task;
	if (!idle)
		panic("No idle process for CPU %d", cpu);

	idle->processor = cpu;
	x86_cpu_to_apicid[cpu] = apicid;
	x86_apicid_to_cpu[apicid] = cpu;
	idle->has_cpu = 1; /* we schedule the first task manually */
	idle->thread.eip = (unsigned long) start_secondary;

	del_from_runqueue(idle);
	unhash_process(idle);
	init_tasks[cpu] = idle;

	/* start_eip had better be page-aligned! */
	start_eip = setup_trampoline();

	/* So we see what's up   */
	printk("Booting processor %d/%d eip %lx\n", cpu, apicid, start_eip);
	stack_start.esp = (void *) (1024 + PAGE_SIZE + (char *)idle);

```

前面讲过，每个 CPU 在运行中必须有自己的上下文，否则就不会对整个系统的运行作出贡献、反而倒会造成损害。所以，必须为每个 CPU 都准备下一个初始进程（线程）。哪怕这个初始进程本身实际上并没有什么事要做，也得要有这么一个进程才能启动这个 CPU。而 CPU 一旦开始了初始进程的运行，以后就可以通过进程调度从系统中挑选其它进程运行了。所以，第一件要做的事就是通过 fork_by_hand()为目标 CPU 创建起一个内核线程。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== arch/i386/kernel/smpboot.c 493 501 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()>fork_by_hand()]

```
static int __init fork_by_hand(void)
{
	struct pt_regs regs;
	/*
	 * don't care about the eip and regs settings since
	 * we'll never reschedule the forked task.
	 */
	return do_fork(CLONE_VM|CLONE_PID, 0, &regs, 0);
}
```

读者对 do_fork()已经不陌生了。这里的第一个参数表示 fork 的是一个线程，共享父进程，即当前进程的用户空间，并且还共享父进程的进程号。第二个参数本来应该是用户空间堆栈的起始地址，第四个参数为堆栈的大小，但是因为创建的是内核线程，所以这两个参数都是 0。寄存器组 regs 的具体内容在这里并没有实际的作用。创建了所需的内核线程以后，do_fork()通过 SET_LINKS()，把它的 task_struct 数据结构链接在就绪进程队列中 init_task 的前面，所以第 560 行可以通过 init_task 取得指向这个 task_struct 数据结构的指针 idle。

代码中的 568 行将 idle->thread.eip 设置成指向 start_secondary()，这是所有的次 CPU，即“应用处理器”，完成了初始化以后进入正常调度时的入口。后面我们会看到它的代码。然后，del_from_runqueue()和 unhash_process()将这个 task_struct 结构从就绪进程队列和杂凑队列中删去，并把它的指针放在数组 init_tasks[]中。这样就只能根据 CPU 序号找到这个 task_struct 结构，而不能像常规的进程那样通过进程号找到它了。读者后面会看到，这个线程就是给定 CPU 的“空转”进程。所有 CPU 的空转进程都具有相同的进程号 0，并且都不挂入就绪进程队列和杂凑队列内，而由数组 init_tasks[]的各个元素指向相应的 task_struct 结构。

函数 start_secondary()的执行有个前提，那就是次 CPU 已经完成了其本身的初始化，已进入了保护模式，并开启了页式存储管理。可是，次 CPU 本身的这些初始化涉及其内部寄存器的操作，因而只能由它自己完成，主 CPU 无法包办代替。再说，当主 CPU 启动一个次 CPU 运行的时候，这个次 CPU 一开始时还处于实地址模式，根本就不能正确地执行 start_secondary()的代码。因此，start_secondary()的作用其实还只是中转性质的，次 CPU 仍然不能“一步登天”进入这个函数中执行，而要先进入一个初始化程序 startup_32()，完成 CPU 本身的初始化（见第 10 章）。可是，即使 startup_32()的执行也还有前提，所以次 CPU 在受到启动之初甚至还不能直接进入初始化的第一阶段 startup_32()，而需要有块“跳板”再来一次中转。读者在第 10 章中将会看到，主 CPU 在进入 startup_32()之前已经在引导辅助程序中进行了一些准备，包括进入保护模式；次 CPU 同样需要这些准备。此外，次 CPU 在进入 startup_32()之前应该把%ebx 的内容设置成 1（而主 CPU 则为 0）。显然，这些准备工作都要在进入 startup_32()之前完成。这就是为什么次 CPU 需要“跳板”而主 CPU 却并不需要的原因。这里 575 行通过 setup_trampoline()为次 CPU 复制好一块跳板，实际上是一段汇编语言程序，并把这段程序的入口地址放在变量 start_eip 中。所以，start_eip 中的地址是启动次 CPU 时的第一站，是低级阶段。而 idle->thread.eip 中的地址则是次 CPU 在完成了自身的初始化，建立起了页面映射以后才开始执行的新起点，是第二站，是高级阶段。函数 setup_trampoline()的代码在 arch/i386/kernel/smpboot.c 中：

==================== arch/i386/kernel/smpboot.c 101 119 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()>setup_trampoline()]

```
/*
 * Trampoline 80x86 program as an array.
 */

extern unsigned char trampoline_data [];
extern unsigned char trampoline_end  [];
static unsigned char *trampoline_base;

/*
 * Currently trivial. Write the real->protected mode
 * bootstrap into the page concerned. The caller
 * has made sure it's suitably aligned.
 */

static unsigned long __init setup_trampoline(void)
{
     memcpy(trampoline_base, trampoline_data, trampoline_end - trampoline_data);
     return virt_to_phys(trampoline_base);
}
```

这段程序是再简单不过的了，但是读者一定会对这“跳板”到底是什么感兴趣。这段代码在 arch/i386/kernel/trampoline.S 中（trampoline 就是跳板的意思）：

==================== arch/i386/kernel/trampoline.S 37 68 ====================

```
ENTRY(trampoline_data)
r_base = .

     mov  %cs, %ax # Code and data in the same place
     mov  %ax, %ds

     mov  $1, %bx       # Flag an SMP trampoline
     cli           # We should be safe anyway

     movl $0xA5A5A5A5, trampoline_data - r_base
                   # write marker for master knows we're running

     lidt idt_48 - r_base    # load idt with 0, 0
     lgdt gdt_48 - r_base    # load gdt with whatever is appropriate

     xor  %ax, %ax
     inc  %ax      # protected mode (PE) bit
     lmsw %ax      # into protected mode
     jmp  flush_instr
flush_instr:
     ljmpl     $__KERNEL_CS, $0x00100000
               # jump to startup_32 in arch/i386/kernel/head.S

```

```
60   idt_48:
61        .word     0              # idt limit = 0
62        .word     0, 0           # idt base = 0L
63
64   gdt_48:
65        .word     0x0800              # gdt limit = 2048, 256 GDT entries
66        .long     gdt_table-__PAGE_OFFSET# gdt base = gdt (first SMP CPU)
67
68   .globl SYMBOL_NAME(trampoline_end)
```

我们在这里不深入解读这段程序了，读者不妨自己加以研究。我们只指出三点：第一，43 行把寄存器%bx 的内容设置成 1，表示这是一个次 CPU。第二，52～54 行将控制寄存器 CR0 的内容设置成 1，就是把 CR0 中的 PE 位（最低位，为 1 表示保护模式）设置成 1，其余位则全都为 0，这样就使 CPU 进入了保护模式，但是页式存储管理则尚未启用（最高位 PG 为 0）。第三，57 行跳转到代码段__KERNEL_CS 中地址为 0x00100000 的地方。读者在第 2 章中看到，常数__KERNEL_CS 的值定义为 0x10，而段描述表 gdt_48 中与此相应的段描述项提供的基地址为 0，所以跳转的目标是 0x00100000。这就是 1MB 处，就是系统在引导以后初始化程序的起点 startup_32()。那正是下一章的重点之一。

读者可能还有个问题：为什么要把这段代码复制到 trampoline_base 中呢？说来话长。当主 CPU 通过 APIC 启动次 CPU 运行时，要把一个启动地址发送给次 CPU。可是，由于 APIC 的内部结构，实际上能发送的只是一个 8 位的物理页面号，称为“向量”。这样，就给启动地址加上了限制：首先，必须与页面边界对齐；其次，必须在 1MB 以下。显然，这里的 trampoline_data()是不满足这两个条件的，所以要另行在 1MB 以下分配一个物理页面，把 trampoline_data()的代码复制到这个页面中，这就是 trampoline_base，是在主 CPU 的初始化阶段通过一个函数 smp_alloc_memory()分配的。

回到 do_boot_cpu()的代码中，579 行把次 CPU 执行 start_secondary()时的堆栈，设置在其 task_struct 数据结构所在的两个页面中（详见第 3 章）。我们继续往下阅读。

==================== arch/i386/kernel/smpboot.c 581 597 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()]

```
581       /*
582        * This grunge runs the startup process for
583        * the targeted processor.
584        */
585
586       atomic_set(&init_deasserted, 0);
587
588       Dprintk("Setting warm reset code and vector.\n");
589
590       CMOS_WRITE(0xa, 0xf);
591       local_flush_tlb();
592       Dprintk("1.\n");
593       *((volatile unsigned short *) phys_to_virt(0x469)) = start_eip >> 4;
594       Dprintk("2.\n");
595       *((volatile unsigned short *) phys_to_virt(0x467)) = start_eip & 0xf;
596       Dprintk("3.\n");
597
```

主 CPU 在启动次 CPU 之前，先要通过其本地 APIC 对次 CPU 的 APIC 执行一次初始化操作，在此期间还把一个全局量 init_deasserted 设成 0，到完成了对目标 APIC 的初始化以后，再把这个变量设置成 1。此外，上述 start_eip 中的入口地址不仅用于本次初始化，还用于系统的“热启动”，所以按照规定将其写入物理地址为 0x467 和 0x469 处。

现在，主 CPU 可以执行给定次 CPU 的启动了。限于篇幅，我们在下面将不深入考察对 APIC 寄存器的操作，有兴趣或有需要的读者可自己加以研究。总而言之，这段代码通过主 CPU 的本地 APIC 中的一些寄存器，包括用来说明发送目标的寄存器 APIC_ICR2 和控制 / 状态寄存器 APIC_ICR，向目标 CPU 的 APIC 发出一些信号，并等待其完成。

==================== arch/i386/kernel/smpboot.c 598 658 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()]

```
	/*
	 * Be paranoid about clearing APIC errors.
	 */
	if (APIC_INTEGRATED(apic_version[apicid])) {
		apic_read_around(APIC_SPIV);
		apic_write(APIC_ESR, 0);
		apic_read(APIC_ESR);
	}

	/*
	 * Status is now clean
	 */
	send_status = 0;
	accept_status = 0;
	boot_status = 0;

	/*
	 * Starting actual IPI sequence...
	 */

	Dprintk("Asserting INIT.\n");

	/*
	 * Turn INIT on target chip
	 */
	apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));

	/*
	 * Send IPI
	 */
	apic_write_around(APIC_ICR, APIC_INT_LEVELTRIG | APIC_INT_ASSERT
				| APIC_DM_INIT);

	Dprintk("Waiting for send to finish...\n");
	timeout = 0;
	do {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
634             Dprintk("+");
635             udelay(100);
636             send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
637         } while (send_status && (timeout++ < 1000));
638
639         mdelay(10);
640
641         Dprintk("Deasserting INIT.\n");
642
643         /* Target chip */
644         apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));
645
646         /* Send IPI */
647         apic_write_around(APIC_ICR, APIC_INT_LEVELTRIG | APIC_DM_INIT);
648
649         Dprintk("Waiting for send to finish...\n");
650         timeout = 0;
651         do {
652             Dprintk("+");
653             udelay(100);
654             send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
655         } while (send_status && (timeout++ < 1000));
656
657         atomic_set(&init_deasserted, 1);
658
```

发出的信号须符合一定的顺序，例如上面的 628 行中先把控制位 APIC_INT_ASSERT 设成 1，然后在下面的 647 行再把它设成 0。每次写入控制 / 状态寄存器 APIC_ICR 以后，都要从这个寄存器读回来，并等待其状态位 APIC_ICR_BUSY 变成 0。不过，等待也不能是无限期的等待，所以还要通过计数加以限制。

==================== arch/i386/kernel/smpboot.c 659 726 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()]

```
        /*
         * Should we send STARTUP IPIs ?
         *
         * Determine this based on the APIC version.
         * If we don't have an integrated APIC, don't
         * send the STARTUP IPIs.
         */
        if (APIC_INTEGRATED(apic_version[apicid]))
            num_starts = 2;
        else
            num_starts = 0;

        /*
         * Run STARTUP IPI loop.
         */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    Dprintk("#startup loops: %d.\n", num_starts);

    maxlvt = get_maxlvt();

    for (j = 1; j <= num_starts; j++) {
        Dprintk("Sending STARTUP #%d.\n",j);
        apic_read_around(APIC_SPIV);
        apic_write(APIC_ESR, 0);
        apic_read(APIC_ESR);
        Dprintk("After apic_write.\n");

        /*
         * STARTUP IPI
         */

        /* Target chip */
        apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));

        /* Boot on the stack */
        /* Kick the second */
        apic_write_around(APIC_ICR, APIC_DM_STARTUP
                       | (start_eip >> 12));

        /*
         * Give the other CPU some time to accept the IPI.
         */
        udelay(300);

        Dprintk("Startup point 1.\n");

        Dprintk("Waiting for send to finish...\n");
        timeout = 0;
        do {
            Dprintk("+");
            udelay(100);
            send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
        } while (send_status && (timeout++ < 1000));

        /*
         * Give the other CPU some time to accept the IPI.
         */
        udelay(200);
        /*
         * Due to the Pentium erratum 3AP.
         */
        if (maxlvt > 3) {
            apic_read_around(APIC_SPIV);
            apic_write(APIC_ESR, 0);
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
723             accept_status = (apic_read(APIC_ESR) & 0xEF);
724             if (send_status || accept_status)
725                 break;
726         }
```

最后，将 start_eip 中的初始化程序入口地址发送给目标 CPU，并等待目标 CPU 的回应。注意这里的 694 和 695 行将 start_eip 中的启动地址写入 APIC 的控制寄存器 APIC_ICR。这个地址是物理地址，在写入寄存器前先右移了 12 位，因为启动地址一定是与页面边界对齐的，其低 12 位一定是 0。另一方面，启动地址一定在 1MB 以下，其最高 12 位也一定是 0，所以右移以后真正要发送出去的是 8 位。

==================== arch/i386/kernel/smpboot.c 727 779 ====================

[start_kernel()>smp_init()>smp_boot_cpus()>do_boot_cpu()]

```
727         Dprintk("After Startup.\n");
728 
729         if (send_status)
730             printk("APIC never delivered???\n");
731         if (accept_status)
732             printk("APIC delivery error (%lx).\n", accept_status);
733 
734         if (!send_status && !accept_status) {
735             /*
736              * allow APs to start initializing.
737              */
738             Dprintk("Before Callout %d.\n", cpu);
739             set_bit(cpu, &cpu_callout_map);
740             Dprintk("After Callout %d.\n", cpu);
741 
742             /*
743              * Wait 5s total for a response
744              */
745             for (timeout = 0; timeout < 50000; timeout++) {
746                 if (test_bit(cpu, &cpu_callin_map))
747                     break;    /* It has booted */
748                 udelay(100);
749             }
750 
751             if (test_bit(cpu, &cpu_callin_map)) {
752                 /* number CPUs logically, starting from 1 (BSP is 0) */
753                 Dprintk("OK.\n");
754                 printk("CPU%d: ", cpu);
755                 print_cpu_info(&cpu_data[cpu]);
756                 Dprintk("CPU has booted.\n");
757             } else {
758                 boot_status = 1;
759                 if (*((volatile unsigned char *)phys_to_virt(8192))
760                         == 0xA5)
761                     /* trampoline started but...? */
762                     printk("Stuck ??\n");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        else
                                /* trampoline code not run */
                                printk("Not responding.\n");
        #if APIC_DEBUG
                        inquire_remote_apic(apicid);
        #endif
                }
        }
        if (send_status || accept_status || boot_status) {
                x86_cpu_to_apicid[cpu] = -1;
                x86_apicid_to_cpu[apicid] = -1;
                cpucount--;
        }

        /* mark "stuck" area as not stuck */
        *((volatile unsigned long *)phys_to_virt(8192)) = 0;
}
```

次 CPU 受到启动以后，首先进入上述的“跳板”，从那里起跳，进入 startup_32()，即初始化的第一阶段（详见第 10 章）。在 startup_32()中，由于次 CPU 的寄存器%ebx 中的内容为 1，就可以跟主 CPU 区分开来，因为有些操作是共同的，而又有些则只由主 CPU 完成。完成了 CPU 本身初始化以后，次 CPU 就通过函数调用进入 initialize_secondary()（见第 10 章，代码在 arch/i386/kernel/smpboot.c 中）：

==================== arch/i386/kernel/smpboot.c 468 486 ====================
[startup_32()>initialize_secondary()]

```
468 /*
469  * Everything has been set up for the secondary
470  * CPUs - they just need to reload everything
471  * from the task structure
472  * This function must not return.
473  */
474 void __init initialize_secondary(void)
475 {
476      /*
477       * We don't actually need to load the full TSS,
478       * basically just the stack pointer and the eip.
479       */
480
481      asm volatile(
482           "movl %0,%%esp\n\t"
483           "jmp *%1"
484           :
485           :"r" (current->thread.esp),"r" (current->thread.eip));
486 }
```

次 CPU 在 startup_32()中，将自己的堆栈设置在主 CPU 为其准备的地方，从而进入了自己的上下文，这就是主 CPU 为之准备好的空转进程。我们在前面看到，这个进程的 thread.eip 指向 start_secondary()，所以 483 行的 jmp 指令就使 CPU 进入了这个函数。同时，482 行已经重新设置了堆

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

栈指针，所以实际上永远不会从 initialize_secondary()返回了。函数 start_secondary()的代码在 arch/i386/kernel/smpboot.c 中：

==================== arch/i386/kernel/smpboot.c 445 466 ====================

```
/*
 * Activate a secondary processor.
 */
int __init start_secondary(void *unused)
{
	/*
	 * Dont put anything before smp_callin(), SMP
	 * booting is too fragile that we want to limit the
	 * things done here to the most necessary things.
	 */
	cpu_init();
	smp_callin();
	while (!atomic_read(&smp_commenced))
		rep_nop();
	/*
	 * low-memory mappings have been cleared, flush them from
	 * the local TLBs too.
	 */
	local_flush_tlb();

	return cpu_idle();
}
```

对次 CPU 的基本初始化是在 startup_32()中完成的，但是进一步的初始化则放在进入 start_secondary()以后才进行，那就是 cpu_init()。这个函数不仅次 CPU 要调用，主 CPU 也要在初始化的第二阶段中调用，其代码可参看第 10 章中的“系统初始化（第二阶段）”一节。与 startup_32()中的初始化相比，这个函数主要是为进程调度作准备。此外，在这个函数中还将全局的位图 cpu_initialized 中与当前 CPU 对应的标志位设成 1，让主 CPU 知道这个次 CPU 的初始化已经完成了。

执行完 cpu_init()以后，次 CPU 的初始化就完成了。这里还要说明一下，我们现在是在讲述其中一个次 CPU 的运行，一个次 CPU 的上下文。主 CPU 在 smp_boot_cpus()的一个 for 循环中，依次对系统中的各个次 CPU 调用 do_boot_cpu()，启动其运行。每启动一个次 CPU，这个 CPU 就会进入 start_secondary()。但是，主 CPU 和次 CPU 之间需要一些通信手段来建立起若干同步点，以取得互相的同步和协调。例如，主 CPU 需要确认刚被启动的次 CPU 已经到达 start_secondary()以后，才能启动下一个次 CPU。而且，不仅主 CPU 和个别的次 CPU 之间需要同步，所有的次 CPU 最后也要步调一致，基本上在同一时间进入 cpu_idle()。具体的同步有下面这么一些：

(1) 全局量 init_deasserted。在 do_boot_cpu()中，主 CPU 在对次 CPU 的 APIC 进行初始化前，先将 init_deasserted 设成 0（586 行），完成了以后再将这个变量设成 1（657 行）。次 CPU 受到启动、通过 start_secondary()进入 smp_callin()以后，就在一个 while 循环中等待这个变量成为 1，从而保证次 CPU 不会在此期间进行对 APIC 的操作。

(2) 全局量位图 cpu_callout_map。次 CPU 在 smp_callin()中还要等待这个位图中的对应位变成 1

PDF-XCHANGE Click to buy NOW! www.docu-track.com

（最多等待 2 秒）。主 CPU 则在 do_boot_cpu()中（739 行）将目标 CPU 的对应位设成 1。

(3) 全局量位图 cpu_callin_map。主 CPU 在 do_boot_cpu()中向次 CPU 发出启动命令以后，就在一个定时的循环中测试目标 CPU 在这个位图中的对应位，等待来自目标 CPU 的回答（745～749 行）。次 CPU 则在 smp_callin()结束之前将位图中的对应位设置成 1。主 CPU 只有测试到这一位变成 1，或者超过了预定的时间，才从 do_boot_cpu()返回。

(4) 全局量 smp_commenced。所有次 CPU 在进入 cpu_init()之前都要等待这个变量成为 1，这个变量将使所有的次 CPU 都站到了同一条起跑线上。

函数 smp_callin()的代码在 arch/i386/kernel/smpboot.c 中：

==================== arch/i386/kernel/smpboot.c 349 439 ====================

[start_secondary()>smp_callin()]

```
void __init smp_callin(void)
{
	int cpuid, phys_id;
	unsigned long timeout;

	/*
	 * If waken up by an INIT in an 82489DX configuration
	 * we may get here before an INIT-deassert IPI reaches
	 * our local APIC.  We have to wait for the IPI or we'll
	 * lock up on an APIC access.
	 */
	while (!atomic_read(&init_deasserted));

	/*
	 * (This works even if the APIC is not enabled.)
	 */
	phys_id = GET_APIC_ID(apic_read(APIC_ID));
	cpuid = current->processor;
	if (test_and_set_bit(cpuid, &cpu_online_map)) {
		printk("huh, phys CPU#%d, CPU#%d already present??\n",
					phys_id, cpuid);
		BUG();
	}
	Dprintk("CPU#%d (phys ID: %d) waiting for CALLOUT\n", cpuid, phys_id);

	/*
	 * STARTUP IPIs are fragile beasts as they might sometimes
	 * trigger some glue motherboard logic. Complete APIC bus
	 * silence for 1 second, this overestimates the time the
	 * boot CPU is spending to send the up to 2 STARTUP IPIs
	 * by a factor of two. This should be enough.
	 */

	/*
	 * Waiting 2s total for startup (udelay is not yet working)
	 */
```

```
	timeout = jiffies + 2*HZ;
	while (time_before(jiffies, timeout)) {
		/*
		 * Has the boot CPU finished it's STARTUP sequence?
		 */
		if (test_bit(cpuid, &cpu_callout_map))
			break;
	}

	if (!time_before(jiffies, timeout)) {
		printk("BUG: CPU%d started up but did not get a callout!\n",
			cpuid);
		BUG();
	}

	/*
	 * the boot CPU has finished the init stage and is spinning
	 * on callin_map until we finish. We are free to set up this
	 * CPU, first the APIC. (this is probably redundant on most
	 * boards)
	 */

	Dprintk("CALLIN, before setup_local_APIC().\n");
	setup_local_APIC();

	sti();

#ifdef CONFIG_MTRR
	/*
	 * Must be done before calibration delay is computed
	 */
	mtrr_init_secondary_cpu ();
#endif
	/*
	 * Get our bogomips.
	 */
	calibrate_delay();
	Dprintk("Stack at about %p\n",&cpuid);

	/*
	 * Save our processor parameters
	 */
	smp_store_cpu_info(cpuid);

	/*
	 * Allow the master to continue.
	 */
	set_bit(cpuid, &cpu_callin_map);

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
434        /*
435         *      Synchronize the TSC with the BP
436         */
437        if (cpu_has_tsc)
438             synchronize_tsc_ap();
439   }
```

将这段代码与前面 smp_boot_cpus()开头处的代码作一比较，就可以看出这里进行的一些操作与主 CPU 在启动次 CPU 之前的操作相似，实际上也是 CPU 初始化的一部分。这里第 421 行调用 calibrate_delay()测试 CPU 的运算速度，主 CPU 则在初始化的第二阶段在 start_kernel()中调用这个函数（见第 10 章）。这个函数测算出 CPU 的“ BogoMIPS”，大致上反映了以“每秒百万条指令”为单位的运算速度。有兴趣的读者不妨自己读一下。

至此，次 CPU 已经基本上完成了初始化，所以在 432 行把全局量 cpu_callin_map 中代表着本 CPU 的标志位设成 1，让主 CPU 知道。最后，如果次 CPU 带有“时间印记计数器 ” TSC 则还要通过 synchronize_tsc_ap()对其 TSC 进行初始化。所谓 TSC 是“ Time Stamp Count”的缩写，这是一个 64 位的计数器，每来一个时钟脉冲就加 1。由于是 64 位计数器，其数值十年以内不会重复，所以可以用它的值作为时间印记。

各个次 CPU 在完成了 smp_callin()的执行以后，就进入了一个无限循环，等待一个全局量 smp_commenced 变成 1，就好像等待“起跑”命令一样，要等到了命令才起跑进入 cpu_idle()。

回到主 CPU 的运行中。主 CPU 在 934～954 行的 for 循环中启动了所有的次 CPU 以后，最后还有几件事要做，我们继续往下看（arch/i386/kernel/smpboot.c）。

==================== arch/i386/kernel/smpboot.c 956 959 ====================

[start_kernel()>smp_init()>smp_boot_cpus()]

```
956        /*
957         * Cleanup possible dangling ends...
958         */
959   #ifndef CONFIG_VISWS
```

==================== arch/i386/kernel/smpboot.c 974 1022 ====================

```
974   #endif
975
976        /*
977         * Allow the user to impress friends.
978         */
979
980        Dprintk("Before bogomips.\n");
981        if (!cpucount) {
982             printk(KERN_ERR "Error: only one processor found.\n");
983        } else {
984             unsigned long bogosum = 0;
985             for (cpu = 0; cpu < NR_CPUS; cpu++)
986                  if (cpu_online_map & (1<<cpu))
987                       bogosum += cpu_data[cpu].loops_per_jiffy;
988             printk(KERN_INFO "Total of %d processors activated (%lu.%02lu BogoMIPS).\n",
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                cpucount+1,
                bogosum/(500000/HZ),
                (bogosum/(5000/HZ))%100);
            Dprintk("Before bogocount - setting activated=1.\n");
    }
    smp_num_cpus = cpucount + 1;

    if (smp_b_stepping)
        printk(KERN_WARNING "WARNING: SMP operation may be unreliable with B stepping processors.\n");
    Dprintk("Boot done.\n");

#ifndef CONFIG_VISWS
    /*
     * Here we can be sure that there is an IO-APIC in the system. Let's
     * go and set it up:
     */
    if (!skip_ioapic_setup)
        setup_IO_APIC();
#endif

    /*
     * Set up all local APIC timers in the system:
     */
    setup_APIC_clocks();

    /*
     * Synchronize the TSC with the AP
     */
    if (cpu_has_tsc && cpucount)
        synchronize_tsc_bp();

smp_done:
    zap_low_mappings();
}
```

首先是显示整个系统的运算能力，即各个 CPU 运算能力的总和。然后通过 setup_IO_APIC()对外部 APIC 进行初始化，这个函数的代码在 arch/i386/kernel/io_apic.c 中，我们只能把它留给读者。接着是对各内部 APIC 中定时器，即时钟中断源的设置，读者已经在前面看过这个函数的代码。

最后，通过 zap_low_mappings()把页面映射目录中低区，即 3GB 以下的目录项都清除掉。这些目录项是为 CPU 由段式映射向页式映射过渡而设置的，详见第 10 章对系统初始化第一阶段的叙述。对于单 CPU 的系统，一旦 CPU 转入页式映射，这些目录项就可以清除了。可是在 SMP 结构的系统中，则要到所有的 CPU 都转入页式映射以后才能清除。而现在是时候了。清除了低区映射以后，swapper_pg_dir 中就只剩下系统空间的映射，所以这个页面映射目录将用于所有的内核线程。同时，这个目录也是所有进程的映射目录的基础，进程的映射目录的系统空间部分就是从 swapper_pg_dir 复制而来。

当主 CPU 从 smp_boot_cpus()返回到 smp_init()中时，所有的次 CPU 都已启动而先后到达同步点，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

已在等待最后的起跑命令了。于是，主 CPU 通过 smp_commence()把全局量 smp_commenced 设置成 1，向所有的次 CPU 发出的起跑命令。这个函数的代码在 arch/i386/kernel/smpboot.c 中：

==================== arch/i386/kernel/smpboot.c 164 185 ====================
[start_kernel()>smp_init()>smp_commence()]

```
/*
 * Architecture specific routine called by the kernel just before init is
 * fired off. This allows the BP to have everything in order [we hope].
 * At the end of this all the APs will hit the system scheduling and off
 * we go. Each AP will load the system gdt's and jump through the kernel
 * init into idle(). At this point the scheduler will one day take over
 * and give them jobs to do. smp_callin is a standard routine
 * we use to track CPUs as they power up.
 */

static atomic_t smp_commenced = ATOMIC_INIT(0);

void __init smp_commence(void)
{
     /*
      * Lets the callins below out of their loop.
      */
     Dprintk("Setting commenced=1, go go go\n");

     wmb();
     atomic_set(&smp_commenced,1);
}
```

这里的 wmb()是个内存写操作路障。将 smp_commenced 设成 1 以后，在 start_secondary()中待命的各个次 CPU 就随之结束了 457 行的 while 循环而进入 cpu_idle()。表面上这是个函数调用，但是实际上永远不会从那里返回了，因为 cpu_idle()的主体是个无限循环，只要系统中有就绪进程在等待执行，就调度其运行，否则就使 CPU 进入硬件睡眠状态，直至有中断发生时才恢复运行。到那时候，既然发生了中断，就又有机会调度了。我们在下一章中还要回到这个函数代码中。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

第10章　系统引导和初始化

10.1　系统引导过程概述

现代计算机系统的内存都是“挥发性”的：一旦关机断电，存储在内存中的信息、连同操作系统本身的映象就丢失了。所以，必须把操作系统（内核）的映象存储在某种不挥发的介质中，使得开机加电时有一个从不挥发介质装入操作系统、并转入运行的映象过程，这个过程就称为“引导”（bootstrap，或 boot），也称为“自举”。这里，所谓不挥发介质通常是指硬磁盘或软盘，但也可以是 EPROM 或 Flash 存储器，还可以是网络中别的节点。一般，从 EPROM 或 Flash 存储器装入映象是很简单的，因为这些存储器本来就是内存的一部分，访问这些存储器与访问普通的内存空间并无不同。相比之下，从磁盘等外部设备装入操作系统映象就复杂多了。所以一般说的“引导”是指从磁盘上引导。我们在本节中将注意力集中在从硬盘引导，因为这是最为典型的；另一方面，理解了怎样从硬盘引导，也就不难理解怎样从别的介质引导了。

可想而知，要在开机时从不挥发介质装入操作系统的映象，系统就得要在一开机时就具有一定的智能，也就是一开机以后 CPU 就能执行一段程序，这段程序本身必须存储在作为系统内存一部分的 EPROM、Flash 等不挥发存储器中，而且必须知道怎样才能从不挥发介质装入操作系统的映象。实际上，各种 CPU 都设计成一加电以后就从某个特殊的地址开始执行指令，所以这些不挥发存储器就放在这个位置上。以 i386 CPU 为例，加电或“总清”（reset）以后 CPU 处于实地址模式，并且代码段寄存器 CS 的内容为 0xffff，而取指令指针（寄存器）IP 的内容则为 0；也就是说，从线性地址 0xffff0 开始取第一条指令。所以采用 i386 CPU 的系统在这个位置上必须有不挥发存储器。

那么，这段程序要有多大呢？这就要看具体的设计了。在早期的计算机中这段程序一般都很小，例如 2K 字节或者更小，甚至只有几条指令（记得 70 年代中美建交后进入中国的 NOVA 机，由此而来的国产机名为 DJS-130，操作系统为 RTOS，引导程序只有 13 条指令，当时称初始引导 13 条，亦称手拨 13 条。13 条指令执行结果是通过光电读入机把存放在穿孔纸带上的 RTOS 执行码装入内存）。这是因为早期的 EPROM 或 PROM 的容量都很小，并且其目的和功能也很单一。可是，这么短的一段程序怎么能把操作系统的映象从磁盘上读进来呢？我们不妨这样想：如果要把操作系统的映象作为一个文件读进来，那么这段程序就要能支持相应的文件系统，还要加上相应的设备驱动程序。一般而言，除十分简单的文件系统以外，这是不现实的。再说，文件系统又有许多种，无论如何也不能使一小段程序同时支持好几种文件系统。所以，只能绕过文件系统，甚至绕过设备驱动的一般形式，而把操作系统映象的读入作为一个特例处理，在设备驱动的底层解决问题。例如，要是从一开始就知道操作系统的映象肯定在磁盘上开头 10 个连续的扇区中，这就比较简单了。如果不能肯定操作系统的映象一定是 10 个扇区，而且一定是前 10 个连续的扇区，那就得在磁盘上某个固定的位置（例如某一个扇区）提供有关操作系统映象的一些“地理”信息。实际上，光是提供这些“地理”信息还是不够的，因为不同文件系统所需要的信息不同，对这些信息的运用也不同。例如，在有些文件系统中，属于同一文件的

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

所有扇区都连成一条链，所以只需提供文件中第一个扇区的位置就行了；而有些文件系统则要采用位图的方式，或者多层映射表的方式；还有，对映象还可能实施了压缩，因而需要在装入映象的同时实施解压缩；而不同 CPU 的指令系统也不同；等等，等等，不一而足。所以，除关于映象的“地理”信息以外，还需要在这个固定的位置上存放一段适用于具体操作系统的可执行程序作为补充。套用现代“面向对象的程序设计”的术语，就是要通过磁盘上的某个固定的位置，通常是一个扇区，提供一个“对象”，其中既包括了数据，也包括了运用这些数据的程序。当然，那时候还没有“对象”这么个概念，而是很自然地把这个扇区称为“引导扇区”，通常都是磁盘上的第一个扇区。而 EPROM 中的程序，则因为其作用在于从磁盘读入引导扇区，因而常常称为初始引导程序。初始引导程序是“中性”的，与具体的操作系统或文件格式无关，它只是从磁盘上读入引导扇区，然后把引导扇区的开头当作一段程序的起点，使 CPU 转入这段程序。引导扇区的内容则取决于具体的操作系统，也可能还进一步取决于文件格式。当然，引导扇区只是一个扇区，即 512 字节，能够容纳的信息和代码是很有限的，所以常常还得要由引导扇区的程序先装入其它若干扇区，再由这些扇区中的程序和数据协同完成整个引导过程。

有时候，也可以先通过引导扇区读入一个作为中间步骤的工具性程序（而不是操作系统映象），常称为“引导装入程序”，再由引导装入程序装入操作系统映象。例如 LILO 就是 Linux 的引导装入程序。从概念上说引导装入程序与 EPROM 中的初始引导程序并无不同，只是体积更大，功能更为复杂。例如 LILO 就可以让用户从磁盘上的多个（并且可以是多种）操作系统映象中有选择地引导。另一方面，由于引导装入程序的功能较强，也有利于减小对操作系统映象在磁盘上存储位置和方式的限制。实际上，除了从软盘引导以外，Linux 一般都是通过 LILO 引导的。

随着技术的发展，像 EPROM 一类存储器件的容量愈来愈大，为加强系统初始引导程序提供了条件，所以逐渐往里面加入了如“加电自检”、“系统配置”一类的功能，（初始的）人机交互界面也逐渐得到改善。到 Microsoft 为 IBM PC 机设计 DOS 操作系统（DOS 是“磁盘操作系统”的缩写）时，更是把 DOS 的整个设备驱动层都放在了 EPROM 中，称为“基本输入 / 输出系统”，即 BIOS。以后，这种格局就一直沿用了下来。在 BIOS 中，初始引导只是其功能的一部分，而且只是很小的一部分。现在的 BIOS 甚至已经比当初的整个 DOS 还要大了，可是再大也不可能把所有操作系统的引导都考虑进去，所以还是采用初始化引导扇区的方案，让各种操作系统通过引导扇区进一步提供其自身的引导手段。其实，虽然 BIOS 提供了对各种主要外部设备的基本驱动，但是，对于多进程、多用户、采用保护模式尤其是页式映射的操作系统却未必合适，所以 Linux 内核并不使用 BIOS 所提供的设备驱动，而是绕开 BIOS，从硬件接口（寄存器）和中断响应开始彻底地实现自己的设备驱动层。对于 Linux 来说，BIOS 的作用只不过就是初始引导。如果要说其它还有什么作用的话，那就是加电以后的自检以及向内核提供在此过程中搜集到的一些信息了（有些信息是由用户改置的）。

另一方面，由于硬盘容量的迅速发展，实际使用中常常把一个硬盘划分成若干“分区”，从而把一个物理的硬盘划分成若干个逻辑磁盘。这样一来，每个逻辑磁盘的第一个扇区仍然是引导扇区，分别用于相应逻辑磁盘中的操作系统映象（如果有的话）。但是，这些引导扇区在物理上当然不再是整个硬盘的第一个扇区了。在这种情况下，整个硬盘的第一个扇区不属于任何一个逻辑磁盘，或者说已经超脱于所有这些逻辑磁盘之外，上了一个层次。但是，BIOS 还是把它当作整个硬盘的引导扇区，加电时还是从这个扇区“引导”，所以这个扇区称为“主引导记录块（扇区）”MBR。MBR 中含有关于盘区划分的信息（通过 fdisk 设置），还有一段简短的程序，一共 512 字节。不过，MBR 中的程序并不直接引

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

导操作系统，而是根据盘区划分的信息从一个预定的“活跃”逻辑磁盘中读入其引导扇区，再由这个引导扇区自己采取进一步的行动。所以，可以把 MBR 的作用看成是为 BIOS 中的初始引导程序提供了一种类似于间接寻址的手段。不过，也可以把用来“引导 ” LILO 的程序（连同有关盘区划分的信息）放在 MBR 中，使整个引导过程少转一道弯。（LILO 或者 GRUB 可以安装在 MBR 或者/hda1 等之中）

可想而知，引导扇区中的程序及其辅助程序（不包括 LILO）必须是很精练的，所以都采用汇编语言编写。这些源代码都在 arch/下面具体 CPU 名下的 boot 目录中。就 i386 而言，都在 arch/i386/boot 中。这个目录中有三个汇编语言的程序：

(1) bootsect.S，这是 Linux 引导扇区的源代码，汇编以后不得超过 512 字节。

(2) sctup.S，这是辅助程序的一部分。

(3) video.S，这是辅助程序的另一部分，用于引导过程中的屏幕显示。

在 arch/i386/boot 中还有个子目录 compressed，含有两个源代码文件 head.S 以及 misc.c。版本较新的内核映象都是经过址缩的，在引导的过程中要解压缩，这两个文件（还有 lib/inflate.c）中的代码就是用于解压缩，也属于辅助程序的一部分。这样，经过编译、汇编、连接以后就形成三个组成部分，即引导扇区的映象 bootsect、辅助程序 setup 以及内核映象本身（通常是 vmlinux）。

严格说来，bootsect 和 setup 并不是内核的一部分，它们的代码又都是用汇编语言编写的，篇幅很大，所以我们在这里只作一些简短的说明，而不深入到这些代码中去了，有兴趣或需要的读者可以自己阅读。这些代码都有比较好的注释，读起来不会太困难。

引导扇区代码 arch/i386/boot/bootsect.S 的作者在文件开头处的注释中，对开始执行这段代码以后的过程作了说明：

==================== arch/i386/boot/bootsect.S 1 26 ====================

```
/*
 *   bootsect.S          Copyright (C) 1991, 1992 Linus Torvalds
 *
 *   modified by Drew Eckhardt
 *   modified by Bruce Evans (bde)
 *   modified by Chris Noe (May 1999) (as86 -> gas)
 *
 * bootsect is loaded at 0x7c00 by the bios-startup routines, and moves
 * itself out of the way to address 0x90000, and jumps there.
 *
 * bde - should not jump blindly, there may be systems with only 512K low
 * memory.  Use int 0x12 to get the top of memory, etc.
 *
 * It then loads 'setup' directly after itself (0x90200), and the system
 * at 0x10000, using BIOS interrupts.
 *
 * NOTE! currently system is at most (8*65536-4096) bytes long. This should
 * be no problem, even in the future. I want to keep it simple. This 508 kB
 * kernel size should be enough, especially as this doesn't contain the
 * buffer cache as in minix (and especially now that the kernel is
 * compressed :-)
 *
 * The loader has been made as simple as possible, and continuous
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * read errors will result in a unbreakable loop. Reboot by hand. It
 * loads pretty fast by getting whole tracks at a time whenever possible.
 */
```

读者大概还是不甚了了，我们再作一些解释。

在 PC 的系统结构中，线性地址 0xA0000 以上，即 640KB 以上都用于图形接口卡以及 BIOS 本身，而 0xA0000 以下的 640KB 为系统的基本内存。如果配备更多的内存，则从 0x100000，即 1MB 处开始，称为“高内存”。当 BIOS（或 LILO）“引导”一个系统时，总是把引导扇区读入到基本内存中地址为 0x7c00 的地方，然后就跳转到 0x7c00 开始执行引导扇区的代码。这段代码将其自身“搬运”到 0x90000 处，并且跳转到那里继续执行，然后通过 BIOS 提供的读磁盘调用“ int 0x13”从磁盘上读入 setup 和内核的映象。其中 setup 的映象读入到地址为 0x90200 的地方，就是经过“搬运”后 bootsect 所在处的上方。然后，就跳转到 setup 的代码中，作好执行内核映象的准备，包括映象的解压缩（如果需要的话），从 BIOS 收集一些数据，在控制台上显示一些信息等等。从 0x90000 到 0xA0000 一共是 64KB，bootsect 只占 512 字节，所以 setup 的大小理论上可达 63.5KB（实际上当然不会那么大）。（当然这种限制也可以修改）

基本内存中开头一部分空间是保留给 BIOS 自己用的，另一方面对于 Linux 内核的引导也需要保留一些运行空间，所以一共保留了 64KB。这样，基本内存中剩下来可用于内核映象的就是 8 个 64KB，即 512KB。但是，在这 512KB 的顶端还要留下 4KB 用于引导命令行（LILO 支持引导命令行）以及一些需要传递给内核的数据（从 BIOS 收集得到）。于是，基本内存中实际可用于内核映象的空间就是 508KB。内核映象一般都是经过压缩的，压缩以后的内核映象就像是一大块数据，跟引导扇区和引导辅助程序的映象拼接在一起，成为内核的“引导映象”。目录 arch/i386/boot/tools 中的 build.c 经编译 / 连接以后产生的可执行程序 build，就是用来拼接引导映象的工具。大小不超过 508KB 的引导映象称为“小映象”，文件名为 zImage；否则就称为“大内核”，文件名为 bzImage。由于 bzImage 在基本内存中已经装载不下，所以要装载在地址为 0x100000（1MB）的地方。不过，不管是 zImage 还是 bzImage，解除压缩以后的内核映象总是放在地址为 0x100000（1MB）的地方。

CPU 在跳转到 bootsect 时尚处于 16 位实地址模式，然后在 setup 的执行过程中转入 32 位保护模式的段式寻址方式。在 bootsect 和 setup 的执行中，二者都利用 BIOS 提供的调用来完成一些比较大的操作，如读磁盘，取得 BIOS 在加电自检时搜集到的有关内存的信息等等。一旦转入内核映象本身的执行，就与 BIOS 分道扬镳，不再使用 BIOS 调用了。

辅助程序 setup 为内核映象的执行作好了准备（包括解除压缩）以后，就跳转到 0x100000 开始内核本身的执行，此后就是内核的初始化过程了。内核的初始化是个非常漫长的过程，整个过程可以分成三个阶段。第一个阶段主要是 CPU 本身的初始化，例如页式映射的建立；第二个阶段主要是系统中一些基础设施的初始化，例如内存管理和进程管理的建立和初始化；最后则是“上层建筑”的初始化，如根设备的安装和外部设备的初始化等等。由于整个过程涉及的代码太大，我们相应地把它分成三节加以叙述。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

10.2　系统初始化（第一阶段）

将 Linux 内核的映象装入内存中，并且作好了一些必要的准备以后，CPU 就通过一条长程转移指令转到映象代码段开头的入口 startup_32，从那里开始执行。对于 SMP 结构的系统，这个时候在运行的只是其中的一个处理器，就是所谓“主 CPU”（也称“引导处理器 ” BP），而其它的“次 CPU”（也称“应用处理器 ” AP）则处于停机状态，等待主 CPU 的启动。次 CPU 在受到启动进入内核时同样也要从 startup_32 开始执行，所以从 startup_32 开始的代码是公共的。但是有些操作仅由主 CPU 执行，另有一些操作则仅由次 CPU 执行。主 CPU 在进入 startup_32 时其寄存器%bx 的内容为 0，而次 CPU 在进入 startup_32 时其寄存器%bx 的内容为 1，在执行的过程中就据此区分执行者（CPU）在系统中扮演的角色。不过，这里要着重指出，这里的代码虽然是公共的，但是并不意味着主 CPU 和次 CPU 有可能并发地执行这段程序。实际上，主 CPU 是开路先锋，它首先执行这段程序，“逢山开路，遇水搭桥”，完成以后才逐个地启动次 CPU 执行，并且等待其完成。所以，在同一时间中，系统中最多只有一个处理器在执行这段程序。

内核映象的起点是 stext，也是_stext，引导和解压缩以后的整个映象放在内存中从 0x100000 即 1MB 开始的区间。CPU 执行内核映象的入口 startup_32 就在内核映象开头的地方，因此其物理地址也是 0x100000。然而，读者在第 2 章中曾看到，在正常运行时整个内核映象都应该在系统空间中，系统空间的地址映射是线性的、连续的，虚拟地址与物理地址间有个固定的位移，这就是 0xC0000000，即 3GB。所以，在连接内核映象时已经在所有的符号地址上加了一个偏移量 0xC0000000，这样 startup_32 的虚拟地址就成了 0xC0100000。

不管是主 CPU 还是次 CPU，进入 startup_32 时都运行于保护模式下的段式寻址方式。段描述表中与__KERNEL_CS 和__KERNEL_DS 相对应的描述项所提供的基地址都是 0，所以实际产生的就是第 2 章中讲过的“线性地址”。其中代码段寄存器 CS 已在进入 startup_32 之前设置成__KERNEL_CS，数据段寄存器则尚未设置成__KERNEL_DS。不过，虽然代码段寄存器已经设置成__KERNEL_CS，从而 startup_32 的地址为 0xC0100000。但是在转入这个入口时使用的指令是“ ljmp 0x100000” 而不是“ ljmp startup_32”，所以装入 CPU 中寄存器 IP 的地址是物理地址 0x100000 而不是虚拟地址 0xC0100000。这样，CPU 在进入 startup_32 以后就会继续以物理地址取指令。只要不在代码段中引用某个地址，例如向某个地址作绝对转移，或者调用某个子程序，就可以一直这样运行下去，而与 CS 的内容无关。此外，CPU 的中断已在进入 startup_32 之前关闭。

从 startup_32 开始的汇编代码在 arch/i386/kernel/head.S 中，这就是初始化的第一阶段。

==================== arch/i386/kernel/head.S 39 93 ====================

[startup_32()]

```
/*
 * swapper_pg_dir is the main page directory, address 0x00101000
 *
 * On entry, %esi points to the real-mode code as a 32-bit pointer.
 */
ENTRY(stext)
ENTRY(_stext)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
startup_32:
/*
 * Set segments to known values
 */
     cld
     movl $(__KERNEL_DS),%eax
     movl %eax,%ds
     movl %eax,%es
     movl %eax,%fs
     movl %eax,%gs
#ifdef CONFIG_SMP
     orw %bx,%bx
     jz 1f

/*
 *   New page tables may be in 4Mbyte page mode and may
 *   be using the global pages.
 *
 *   NOTE! If we are on a 486 we may have no cr4 at all!
 *   So we do not try to touch it unless we really have
 *   some bits in it to set.  This won't work if the BSP
 *   implements cr4 but this AP does not -- very unlikely
 *   but be warned!  The same applies to the pse feature
 *   if not equally supported. --macro
 *
 *   NOTE! We have to correct for the fact that we're
 *   not yet offset PAGE_OFFSET..
 */
#define cr4_bits mmu_cr4_features-__PAGE_OFFSET
     cmpl $0,cr4_bits
     je 3f
     movl %cr4,%eax         # Turn on paging options (PSE,PAE,..)
     orl cr4_bits,%eax
     movl %eax,%cr4
     jmp 3f
1:
#endif
/*
 * Initialize page tables
 */
     movl $pg0-__PAGE_OFFSET,%edi /* initialize page tables */
     movl $007,%eax         /* "007" doesn't mean with right to kill, but
                  PRESENT+RW+USER */
2:   stosl
     add $0x1000,%eax
     cmp $empty_zero_page-__PAGE_OFFSET,%edi
     jne 2b

```

首先将 CPU 中除 CS 以外所有的段寄存器，即%ds、%es、%fs 以及%gs，都设置成__KERNEL_DS，读者在第 2 章中看到过这个数值是 0x18，表示采用全局段描述表 GDT 中下标为 3 的段描述项，而 RPL 则为 0。引导辅助程序在转入 startup_32 之前准备下了一个临时的全局段描述表，表中相应的描述项给出基地址为 0，也就是使用线性地址。

不管是主 CPU 还是次 CPU，在 Linux 内核中都要进入页式映射模式运行。暂时以物理地址取指令虽然是可以的，但是不能在代码中向符号地址作绝对转移或调用子程序，因此终非长久之计，所以，要尽快准备好页面映射表并开启 CPU 的页面映射机制。代码中的 86～92 行将从 pg0 开始直到 empty_zero_page 之间的 8K 字节设置成一个临时的页面映射表。这个页面表在内存中，是由所有 CPU 公用的，所以只由主 CPU 进行初始化，次 CPU 则要跳过这一小段代码（见 57～58 行以及 76 和 80 行）。此外，对于次 CPU 这里还有个小插曲，那就是如果系统支持 PSE/PAE，即 36 位地址模式的话还要相应地设置其控制寄存器%cr4（77～79 行）。

==================== arch/i386/kernel/head.S 399 415 ====================

```
399  /*
400   * The page tables are initialized to only 8MB here - the final page
401   * tables are set up later depending on memory size.
402   */
403  .org 0x2000
404  ENTRY(pg0)
405
406  .org 0x3000
407  ENTRY(pg1)
408
409  /*
410   * empty_zero_page must immediately follow the page tables ! (The
411   * initialization loop counts until empty_zero_page)
412   */
413
414  .org 0x4000
415  ENTRY(empty_zero_page)
```

常数 __PAGE_OFFSET 为系统空间的虚拟地址与物理地址之间的差距，定义于 include/asm-i386/page.h：

==================== include/asm-i386/page.h 81 81 ====================

```
81   #define __PAGE_OFFSET       (0xC0000000)
```

可见，pg0 在地址（相对于程序的起点，见下面）为 0x2000 的地方。这个页面映射表中的表项依次设置为 0x7、0x1007、0x2007 等等。其中最低的三位均为 1，表示页面为用户页面，可写，并且页面的内容在内存中（参阅第 2 章）。映射的目标，即物理页面的基地址，则分别为 0x0、0x1000、0x2000 等等，也就是物理内存中的页面 0、1、2 等等。映射表的大小是两个页面，即 2K 个表项，所以代表着一块 8MB 的存储空间，这就是 Linux 内核对内存大小的最低限度要求。

那么，页面目录在哪儿呢？

==================== arch/i386/kernel/head.S 383 397 ====================

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
383 /*
384  * This is initialized to create an identity-mapping at 0-8M (for bootup
385  * purposes) and another mapping of the 0-8M area at virtual address
386  * PAGE_OFFSET.
387  */
388 .org 0x1000
389 ENTRY(swapper_pg_dir)
390      .long 0x00102007
391      .long 0x00103007
392      .fill BOOT_USER_PGD_PTRS-2,4,0
393      /* default: 766 entries */
394      .long 0x00102007
395      .long 0x00103007
396      /* default: 254 entries */
397      .fill BOOT_KERNEL_PGD_PTRS-2,4,0
```

这里的 392 行利用汇编语言提供的功能在其所在的位置上填入 766 个目录项，每个目录项的大小为 4 个字节，内容为 0。同样，397 行也填入 254 个这样的目录项。这里引用的几个常数均定义于 include/asm-i386/pgtable.h：

==================== include/asm-i386/pgtable.h 126 128 ====================

```
#define TWOLEVEL_PGDIR_SHIFT    22
#define BOOT_USER_PGD_PTRS (__PAGE_OFFSET >> TWOLEVEL_PGDIR_SHIFT)
#define BOOT_KERNEL_PGD_PTRS (1024-BOOT_USER_PGD_PTRS)
```

回顾一下第 2 章的内容，这里的常数 __PAGE_OFFSET 是 0xC0000000，所以 BOOT_USER_PGD_PTRS 为 768，而 BOOT_KERNEL_PGD_PTRS 则为 256。一个页面目录的大小是 4KB，含有 1024 个目录项，共代表着 4GB 的虚存空间。Linux 内核以 3GB 为界把整个虚存空间分成用户空间和系统空间两部分。所以，页面目录中的低 768 个目录项用于用户空间的映射，而高 256 个目录项用于系统空间的映射。在初始的页面目录 swapper_pg_dir 中，用户空间和系统空间都只映射了开头的两个目录项（390～391 行以及 394～395 行），即 8MB 的空间，而且有着相同的映射，即指向相同的页面映射表。这样，以符号地址 empty_zero_page 为例，在连接内核映象时这个地址显然是安排在 0xC0000000 以上的空间，而物理上却是装入在(empty_zero_page - 0xC0000000)的地址上。所以，若在未开启页式映射之前要引用这个符号，就要从中减去位移量__PAGE_OFFSET（见 86 行和 91 行）。可是，一旦开启页式映射以后，再要引用这个符号时就可以（而且应该）直接引用了。再来看目录项的内容，以 394 行为例，这个目录项指向物理地址 0x00102000，这是在 1MB 以上的区间中，实际上就是 pg0 的物理地址。前面 403 和 404 行指定 pg0 的起点地址为 0x2000，那是假定整个内核映象是从地址 0 开始时的起点，而现在内核映象放在从地址 0x00100000 开始的地方，pg0 的起点也就相应地变成了 0x00102000。

那么，为什么在虚存空间的低区，即本来应该属于用户空间的位置上（390～391 行）也要放上同样的目录项呢？简而言之是为了平稳过渡。如上所述，CPU 进入 startup_32()以后是以物理地址来取指令的。在这种情况下，如果映射目录只包括系统空间，即虚存空间高区的映射，而不包括虚存空间低区的映射，则一旦开启页式映射以后就不能继续执行了，因为此时 CPU 中的取指令指针 IP 仍指向低

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

区，仍会以物理地址取指令，直到以某个符号地址为目标作绝对转移或调用子程序时为止。所以，解决的办法是分两步走：

(1) 先开启页式映射，但是在虚存空间的低区暂时提供与高区相同的映射，使 CPU 可以继续执行。
(2) 在开启了页式映射之后，以一个符号地址为目标执行一条绝对转移指令。如前所述，由符号所代表的地址是系统空间的虚拟地址。CPU 在执行绝对转移指令时把这个虚拟地址装入 IP，从此就改成以虚拟地址在系统空间取指令了。

CPU 转入系统空间以后，应该把低区的映射清除。后面读者将会看到，页面映射目录 swapper_pg_dir 经过扩充以后就成为所有内核线程的页面映射目录。在内核线程的正常运行中，处于系统状态的 CPU 是不应该通过用户空间的虚拟地址访问内存的。清除了低区的映射以后，如果发生 CPU 在内核中通过用户空间的虚拟地址访问内存，就可以因为产生页面异常而抓住这个错误。

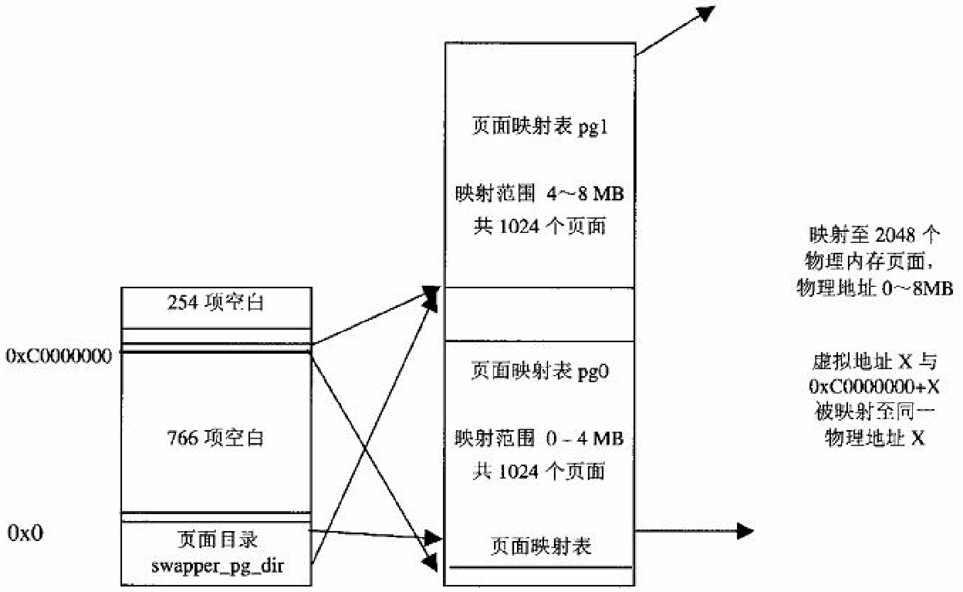


图 10.1　初始化第一阶段的页面映射

在开启页式映射完成向系统空间过渡之前，如果必须要作绝对转移或系统调用，就得在目标地址上减去位移量__PAGE_OFFSET，即 0xC0000000。就凭这一点，向页式映射和系统空间的过渡就宜早不宜迟。所以，现在是开启页面映射机制的时候了。注意下面又是主 CPU 和次 CPU 共用的代码了。页面映射是 CPU 内部的功能，每个 CPU 都要开启其自己的页面映射机制，尽管使用的页面映射目录和页面映射表是相同的。

==================== arch/i386/kernel/head.S 94 110 ====================
[startup_32()]

```
/*
 * Enable paging
 */
3:
	movl $swapper_pg_dir-__PAGE_OFFSET,%eax
```

```
99        movl %eax,%cr3          /* set the page table pointer.. */
100       movl %cr0,%eax
101       orl $0x80000000,%eax
102       movl %eax,%cr0          /* ..and set paging (PG) bit */
103       jmp 1f             /* flush the prefetch-queue */
104  1:
105       movl $1f,%eax
106       jmp *%eax     /* make sure eip is relocated */
107  1:
108       /* Set up the stack pointer */
109       lss stack_start,%esp
110
```

将页面目录的地址装入控制寄存器%cr3，并把%cr0 中的最高位设置成 1，就开启了 CPU 的页面映射机制。装入控制寄存器%cr3 的地址必须是页面目录的物理地址，所以要从虚拟地址 swapper_pg_dir 中减去偏移量，把它还原为物理地址。

开启页面映射机制以后立即就有条相对转移指令（103～104 行），从逻辑上说这条指令不起什么作用，但是它起到了丢弃已经在 CPU 的取指令流水线中内容的作用，这是 Intel 在 i386 的技术资料中建议的。由于跳转的目标离得很近，103 行的 jmp 指令是条相对转移指令，只是在 IP 的当前值上加了一个不大的位移，CPU 仍旧以物理地址取指令而并没有转入系统空间，所以至 104 行为止只是完成了上述的第一步。但是，从此开始，通过数据段引用符号地址时就不需要从中减去偏移量__PAGE_OFFSET 了。紧接着的另一条转移指令（106 行）就不同了，105 行把目标地址置入%eax 时是通过数据段引用这个符号的，所以这个地址在系统空间中，然后以%eax 中的这个地址为目标执行 jmp 指令，就使 CPU 转入了系统空间，完成了两种模式间的平稳过渡。

本来，现在可以清除页面映射目录低区的那些目录项了，可是，考虑到次 CPU 将来也要有这么个过渡的过程，所以还得暂时保留着，到系统中所有的 CPU 都完成了过渡才来清除。

代码中的 109 行将 CPU 的堆栈设置在 stack_start 处。

==================== arch/i386/kernel/head.S 330 332 ====================

```
ENTRY(stack_start)
     .long SYMBOL_NAME(init_task_union)+8192
     .long __KERNEL_DS
```

这里的 init_task_union 是个 union，其类型定义于 include/linux/sched.h:

==================== include/linux/sched.h 480 487 ====================

```
#ifndef INIT_TASK_SIZE
# define INIT_TASK_SIZE 2048*sizeof(long)
#endif

union task_union {
     struct task_struct task;
     unsigned long stack[INIT_TASK_SIZE/sizeof(long)];
};
```

也就是说，既可以把 task_union 看成一个 task_struct 结构，也可以把它看成一个数组，数组的大小

是 8K 字节，即两个页面。读者在第 4 章中看到，每个进程的 task_struct 数据结构和系统空间堆栈共占两个页面，下部是 task_struct 结构，上部是系统空间堆栈。可想而知，这里是在为创建系统中的第一个进程进行准备，而具体的数据结构 init_task_union 则定义于 arch/i386/kernel/init_task.c：

==================== arch/i386/kernel/init_task.c 15 24 ====================

```
/*
 * Initial task structure.
 *
 * We need to make sure that this is 8192-byte aligned due to the
 * way process stacks are handled. This is done by having a special
 * "init_task" linker map entry..
 */
union task_union init_task_union
	__attribute__((__section__(".data.init_task"))) =
		{ INIT_TASK(init_task_union.task) };
```

数据结构的内容由宏定义 INIT_TASK()决定，其代码在 include/linux/sched.h 中：

==================== include/linux/sched.h 434 477 ====================

```
/*
 *  INIT_TASK is used to set up the first task table, touch at
 * your own risk!. Base=0, limit=0x1fffff (=2MB)
 */
#define INIT_TASK(tsk) \
{                                           \
    state:          0,                          \
    flags:          0,                          \
    sigpending:         0,                          \
    addr_limit:         KERNEL_DS,                          \
    exec_domain:    &default_exec_domain,                       \
    lock_depth:         -1,                         \
    counter:        DEF_COUNTER,                        \
    nice:       DEF_NICE,                       \
    policy:         SCHED_OTHER,                        \
    mm:             NULL,                           \
    active_mm:      &init_mm,                       \
    cpus_allowed:   -1,                         \
    run_list:       LIST_HEAD_INIT(tsk.run_list),               \
    next_task:      &tsk,                           \
    prev_task:      &tsk,                           \
    p_opptr:        &tsk,                           \
    p_pptr:         &tsk,                           \
    thread_group:   LIST_HEAD_INIT(tsk.thread_group),           \
    wait_chldexit:  __WAIT_QUEUE_HEAD_INITIALIZER(tsk.wait_chldexit),\
    real_timer:         {                           \
      function:     it_real_fn                      \
    },                                  \
    cap_effective:  CAP_INIT_EFF_SET,                       \
    cap_inheritable:    CAP_INIT_INH_SET,                       \
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    cap_permitted:  CAP_FULL_SET,                    \
    keep_capabilities:  0,                           \
    rlim:      INIT_RLIMITS,                     \
    user:      INIT_USER,                        \
    comm:      "swapper",                        \
    thread:        INIT_THREAD,                      \
    fs:            &init_fs,                     \
    files:         &init_files,                      \
    sigmask_lock:  SPIN_LOCK_UNLOCKED,                   \
    sig:       &init_signals,                        \
    pending:       { NULL, &tsk.pending.head, {{0}}},        \
    blocked:       {{0}},                            \
    alloc_lock:        SPIN_LOCK_UNLOCKED                \
}
```

这个进程的 thread 结构定义为全 0：

==================== include/asm-i386/processor.h 379 387 ====================

```
#define INIT_THREAD  {                         \
     0,                                   \
     0, 0, 0, 0,                              \
     { [0 ... 7] = 0 }, /* debugging registers */   \
     0, 0, 0,                             \
     { { 0, }, },       /* 387 state */            \
     0,0,0,0,0,0,                             \
     0,{~0,}            /* io permissions */       \
}
```

显然，这个进程的名称是 “ swapper ”。代码的作者在注释中警告不要轻易改变它的内容。读者还要注意，不要把这个进程与后面讲到的 init 进程相混淆。

读者也许要问，上面 98～109 行这段代码是共同的，如果主 CPU 和次 CPU 都把系统空间堆栈设置在同一个地方，难道就不会引起冲突吗？不会，这两个页面的使用只是暂时的，在同一时间中只可能有一个 CPU 在使用，而且用完了就不会再回来，所以不会造成问题。读者在后面就会看到，次 CPU 在经由 initialize_secondary()进入 start_secondary()的途中会将其系统空间堆栈（从而其 task_struct 结构）更换到由主 CPU 为之准备好的地方。

==================== arch/i386/kernel/head.S 111 135 ====================

[startup_32()]

```
111 #ifdef CONFIG_SMP
112     orw  %bx,%bx
113     jz  1f                  /* Initial CPU cleans BSS */
114     pushl $0
115     popfl
116     jmp checkCPUtype
117 1:
118 #endif CONFIG_SMP
119
120 /*
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
121  * Clear BSS first so that there are no surprises...
122  * No need to cld as DF is already clear from cld above...
123  */
124      xorl %eax,%eax
125      movl $ SYMBOL_NAME(__bss_start),%edi
126      movl $ SYMBOL_NAME(_end),%ecx
127      subl %edi,%ecx
128      rep
129      stosb
130 
131 /*
132  * start system 32-bit setup. We need to re-do some of the things done
133  * in 16-bit mode for the "real" operations.
134  */
135      call setup_idt
```

对于次 CPU，只是在这里的 114～116 行将其“标志寄存器”设置成 0，接着就转到下面的标号 checkCPUtype 处了。而主 CPU 则担任着开路先锋的角色，需要作更大的贡献。

第一件事是初始化内核的 bss 段。内核的映象也跟其它的可执行程序一样有个 bss 段，bss 段中是一些全局变量或者静态（static）变量，需要在开始运行程序的主体之前将这个区间全部清 0。这里的 124～129 行把从__bss_start 开始到_end 为止的 bss 段全部清 0。顺便提一下，像__bss_start、_end 这些符号的值是由 gcc 在编译和连接时自动生成的。

第二件事是通过 setup_idt()设置初始状态的中断向量表，或曰中断描述表。读者在第 3 章中已经看到，每个表项的大小是 8 个字节，共有 256 个表项。函数 setup_idt()的代码在同一个文件中。

==================== arch/i386/kernel/head.S 304 328 ====================

[startup_32()>setup_idt()]

```
/*
 *  setup_idt
 *
 *  sets up a idt with 256 entries pointing to
 *  ignore_int, interrupt gates. It doesn't actually load
 *  idt - that can be done only after paging has been enabled
 *  and the kernel moved to PAGE_OFFSET. Interrupts
 *  are enabled elsewhere, when we can be relatively
 *  sure everything is ok.
 */
setup_idt:
     lea ignore_int,%edx
     movl $(__KERNEL_CS << 16),%eax
     movw %dx,%ax        /* selector = 0x0010 = cs */
     movw $0x8E00,%dx    /* interrupt gate - dpl=0, present */

     lea SYMBOL_NAME(idt_table),%edi
     mov $256,%ecx
rp_sidt:
     movl %eax,(%edi)
```

```
324         movl %edx,4(%edi)
325         addl $8,%edi
326         dec %ecx
327         jne rp_sidt
328         ret
```

初始状态的 256 个中断描述项全都相同,都指向同一个中断响应程序 ignore_int()。代码的 315～318 行在寄存器%edx 和%eax 中构筑起一个中断门的映象，然后就通过一个循环将相同的内容写入从 idt_table 开始的 256 个表项中。读者不妨结合第 3 章中的有关内容搞清楚：这些表项的函数指针指向 ignore_int()，而 P 标志位为 1（表示在内存中），DPL 为 0（级别最高），D 标志位为 1（32 位），类型码为 110（中断门），合在一起就是 0x8e00。此外，读者在第 3 章中已经看到，这些表项的内容以后可以通过 set_trap_gate()、set_system_gate()等函数加以改变。中断响应程序 ignore_int()的代码也在同一文件中。

==================== arch/i386/kernel/head.S 337 356 ====================

```
337         ALIGN
338  ignore_int:
339         cld
340         pushl %eax
341         pushl %ecx
342         pushl %edx
343         pushl %es
344         pushl %ds
345         movl $(__KERNEL_DS),%eax
346         movl %eax,%ds
347         movl %eax,%es
348         pushl $int_msg
349         call SYMBOL_NAME(printk)
350         popl %eax
351         popl %ds
352         popl %es
353         popl %edx
354         popl %ecx
355         popl %eax
356         iret
```

==================== arch/i386/kernel/head.S 334 336 ====================

```
334  /* This is the default interrupt "handler" :-) */
335  int_msg:
336         .asciz "Unknown interrupt\n"
```

就是说，如果发生中断就通过 printk()在屏幕上显示一行出错信息。在初始化期间，printk()在屏幕上显示信息，而在系统转入正式运行以后则一般都是将信息写入系统的运行日志/var/messages。至于中断描述表 idt_table[ ]，则是在 arch/i386/kernel/traps.c 中定义的全局量。

==================== arch/i386/kernel/traps.c 58 63 ====================

```
58   /*
59    * The IDT has to be page-aligned to simplify the Pentium
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * F0 0F bug workaround.. We have a special link segment
 * for this.
 */
struct desc_struct idt_table[256] __attribute__((__section__(".data.idt"))) = { {0, 0}, };
```

不过，这里还只是为这种中断响应作了一些准备，尚未把中断描述表的地址设置进“中断描述表寄存器 ” IDTR，也没有打开中断。

第三件事与引导命令行和参数有关。如前所述，LILO 允许在引导时使用命令行，并将其传递给内核，而 setup 则从 BIOS 收集一些数据，作为“引导参数”一起传递给引导进来的内核。这些数据合在一起占据不超过一个页面的空间，只是在初始化期间才用到。内核中本来有个内容为全 0 的页面 empty_zero_page，代码中常常通过宏定义 ZERO_PAGE 引用的就是这个页面。不过，这个页面要到初始化完成、系统转入正常运行时才会用到，现在不妨先利用一下，所以把命令行和引导参数都复制到这个页面中。这样，这些数据原来占据的页面就腾了出来，可以回收了（而 empty_zero_page 反正是不能回收的）。

==================== arch/i386/kernel/head.S 136 171 ====================

[startup_32()]

```
/*
 * Initialize eflags.  Some BIOS's leave bits like NT set.  This would
 * confuse the debugger if this code is traced.
 * XXX - best to initialize before switching to protected mode.
 */
	pushl $0
	popfl
/*
 * Copy bootup parameters out of the way. First 2kB of
 * _empty_zero_page is for boot parameters, second 2kB
 * is for the command line.
 *
 * Note: %esi still has the pointer to the real-mode data.
 */
	movl $ SYMBOL_NAME(empty_zero_page),%edi
	movl $512,%ecx
	cld
	rep
	movsl
	xorl %eax,%eax
	movl $512,%ecx
	rep
	stosl
	movl SYMBOL_NAME(empty_zero_page)+NEW_CL_POINTER,%esi
	andl %esi,%esi
	jnz 2f			# New command line protocol
	cmpw $(OLD_CL_MAGIC),OLD_CL_MAGIC_ADDR
	jne 1f
	movzwl OLD_CL_OFFSET,%esi
	addl $(OLD_CL_BASE_ADDR),%esi
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
166  2:
167       movl $ SYMBOL_NAME(empty_zero_page)+2048,%edi
168       movl $512,%ecx
169       rep
170       movsl
171  1:
```

将 setup 传递过来的引导参数和命令行复制到 empty_zero page 中以后，就在这个页面中形成了一个“参数块”，其内容如下（见 arch/i386/kernel/setup.c）：

==================== arch/i386/kernel/setup.c 151 172 ====================

```
151  /*
152   * This is set up by the setup-routine at boot-time
153   */
154  #define PARAM ((unsigned char *)empty_zero_page)
155  #define SCREEN_INFO (*(struct screen_info *) (PARAM+0))
156  #define EXT_MEM_K (*(unsigned short *) (PARAM+2))
157  #define ALT_MEM_K (*(unsigned long *) (PARAM+0x1e0))
158  #define E820_MAP_NR (*(char*) (PARAM+E820NR))
159  #define E820_MAP    ((struct e820entry *) (PARAM+E820MAP))
160  #define APM_BIOS_INFO (*(struct apm_bios_info *) (PARAM+0x40))
161  #define DRIVE_INFO (*(struct drive_info_struct *) (PARAM+0x80))
162  #define SYS_DESC_TABLE (*(struct sys_desc_table_struct*)(PARAM+0xa0))
163  #define MOUNT_ROOT_RDONLY (*(unsigned short *) (PARAM+0x1F2))
164  #define RAMDISK_FLAGS (*(unsigned short *) (PARAM+0x1F8))
165  #define ORIG_ROOT_DEV (*(unsigned short *) (PARAM+0x1FC))
166  #define AUX_DEVICE_INFO (*(unsigned char *) (PARAM+0x1FF))
167  #define LOADER_TYPE (*(unsigned char *) (PARAM+0x210))
168  #define KERNEL_START (*(unsigned long *) (PARAM+0x214))
169  #define INITRD_START (*(unsigned long *) (PARAM+0x218))
170  #define INITRD_SIZE (*(unsigned long *) (PARAM+0x21c))
171  #define COMMAND_LINE ((char *) (PARAM+2048))
172  #define COMMAND_LINE_SIZE 256
```

随着代码的阅读，读者自会慢慢明白这些数据的用途。

至此，需要由主 CPU 单独进行的操作，即上述的三件事情都已经完成了。从标号 checkCPUtype 开始，又是所有 CPU 公共的代码了（见 116 行的 jmp 指令）。我们继续往下看。

==================== arch/i386/kernel/head.S 172 246 ====================

[startup_32()]

```
172  #ifdef CONFIG_SMP
173  checkCPUtype:
174  #endif
175
176       movl $-1,X86_CPUID     #  -1 for no CPUID initially
177
178  /* check if it is 486 or 386. */
179  /*
180   * XXX - this does a lot of unnecessary setup.  Alignment checks don't
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * apply at our cpl of 0 and the stack ought to be aligned already, and
 * we don't need to preserve eflags.
 */

	movl $3,X86		# at least 386
	pushfl			# push EFLAGS
	popl %eax	# get EFLAGS
	movl %eax,%ecx		# save original EFLAGS
	xorl $0x40000,%eax # flip AC bit in EFLAGS
	pushl %eax		# copy to EFLAGS
	popfl			# set EFLAGS
	pushfl			# get new EFLAGS
	popl %eax	# put it in eax
	xorl %ecx,%eax		# change in flags
	andl $0x40000,%eax # check if AC bit changed
	je is386

	movl $4,X86		# at least 486
	movl %ecx,%eax
	xorl $0x200000,%eax	# check ID flag
	pushl %eax
	popfl			# if we are on a straight 486DX, SX, or
	pushfl			# 487SX we can't change it
	popl %eax
	xorl %ecx,%eax
	pushl %ecx		# restore original EFLAGS
	popfl
	andl $0x200000,%eax
	je is486

	/* get vendor info */
	xorl %eax,%eax			# call CPUID with 0 -> return vendor ID
	cpuid
	movl %eax,X86_CPUID		# save CPUID level
	movl %ebx,X86_VENDOR_ID		# lo 4 chars
	movl %edx,X86_VENDOR_ID+4	# next 4 chars
	movl %ecx,X86_VENDOR_ID+8	# last 4 chars

	orl %eax,%eax		# do we have processor info as well?
	je is486

	movl $1,%eax		# Use the CPUID instruction to get CPU type
	cpuid
	movb %al,%cl		# save reg for future use
	andb $0x0f,%ah		# mask processor family
	movb %ah,X86
	andb $0xf0,%al		# mask model
	shrb $4,%al
	movb %al,X86_MODEL
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
230       andb $0x0f,%cl         # mask mask revision
231       movb %cl,X86_MASK
232       movl %edx,X86_CAPABILITY
233
234   is486:
235       movl %cr0,%eax         # 486 or better
236       andl $0x80000011,%eax  # Save PG,PE,ET
237       orl $0x50022,%eax  # set AM, WP, NE and MP
238       jmp 2f
239
240   is386:    pushl %ecx         # restore original EFLAGS
241       popfl
242       movl %cr0,%eax         # 386
243       andl $0x80000011,%eax  # Save PG,PE,ET
244       orl $2,%eax        # set MP
245   2:  movl %eax,%cr0
246       call check_x87
```

读者知道，i386 是一个 CPU 芯片系列，其中包括了 80386、80486 以及后来的各种 Pentium 芯片。为了让在具体 CPU 芯片上运行的软件知道是在什么样的 CPU 上运行，在各种 CPU 芯片中都提供了一些手段，让软件可以在运行中加以测试。可是，Intel 并没有从一开始就有一个全盘的考虑，而只是到了 Pentium 才为此专设了一条指令 cpuid。这条指令在寄存器%eax 中返回 CPU 芯片所属的系列、型号、版本等信息。读者在前一章中还看到，这条指令还起着存储器路障的作用。至于 Pentium 以前的处理器，即 80386 和 80486，则要通过一些特殊的操作才能知道是哪一种处理器。我们在这里就不深入到这些细节中去了。至于 246 行，则是测试与 CPU 配套的浮点协处理器 80387 是否存在。

我们在前面讲过，CPU 在进入 startup_32()时已经运行于保护模式。既然运行于保护模式，那就有个全局段描述表，而控制寄存器 GDTR 则指向这个全局段描述表。可是，那时候的全局段描述表只是临时的，由引导辅助程序 setup 设置。至此为止，CPU 一直在用这个临时的全局段描述表。现在，则要改成使用内核正式的全局段描述表了。同样，此前的控制寄存器 IDTR 也指向临时的中断描述表，现在也要转到内核正式的中断描述表了。这些事也是每个 CPU 都要做的。

==================== arch/i386/kernel/head.S 247 266 ====================

[startup_32()]

```
#ifdef CONFIG_SMP
    incb ready
#endif
    lgdt gdt_descr
    lidt idt_descr
    ljmp $(__KERNEL_CS),$1f
1:  movl $(__KERNEL_DS),%eax     # reload all the segment registers
    movl %eax,%ds      # after changing gdt.
    movl %eax,%es
    movl %eax,%fs
    movl %eax,%gs
#ifdef CONFIG_SMP
    movl $(__KERNEL_DS), %eax
```

```
        movl %eax,%ss        # Reload the stack pointer (segment only)
#else
        lss stack_start,%esp    # Load processor stack
#endif
        xorl %eax,%eax
        lldt %ax
        cld             # gcc2 wants the direction flag cleared at all times
```

指令 lgdt 和 lidt 分别设置 CPU 的“全局段描述表寄存器”GDTR 和“中断描述表寄存器”IDTR。实际上装入这些寄存器的是一个 Xgt_desc_struct 数据结构，此数据结构的类型定义见 include/asm-i386/desc.h：

==================== include/asm-i386/desc.h 51 54 ====================

```
struct Xgt_desc_struct {
        unsigned short size;
        unsigned long address __attribute__((packed));
};
```

结构中首先是 16 位的字段 size，表示描述表的大小，然后才是具体描述表的地址。这两个数据结构的空间分配也是在 arch/i386/kernel/head.S 中定义的：

==================== arch/i386/kernel/head.S 372 381 ====================

```
idt_descr:
        .word IDT_ENTRIES*8-1       # idt contains 256 entries
SYMBOL_NAME(idt):
        .long SYMBOL_NAME(idt_table)

        .word 0
gdt_descr:
        .word GDT_ENTRIES*8-1
SYMBOL_NAME(gdt):
        .long SYMBOL_NAME(gdt_table)
```

中断描述表 idt_table 的内容已经在前面设置好，现在又设置了 IDTR，就完成了对中断机制的准备，只差打开中断了。

全局段描述表 gdt_table 的定义也在同一文件中：

==================== arch/i386/kernel/head.S 450 458 ====================

```
ENTRY(gdt_table)
        .quad 0x0000000000000000    /* NULL descriptor */
        .quad 0x0000000000000000    /* not used */
        .quad 0x00cf9a000000ffff    /* 0x10 kernel 4GB code at 0x00000000 */
        .quad 0x00cf92000000ffff    /* 0x18 kernel 4GB data at 0x00000000 */
        .quad 0x00cffa000000ffff    /* 0x23 user   4GB code at 0x00000000 */
        .quad 0x00cff2000000ffff    /* 0x2b user   4GB data at 0x00000000 */
        .quad 0x0000000000000000    /* not used */
        .quad 0x0000000000000000    /* not used */
```

由于改变了 GDTR 的内容，各个段寄存器的内容也要再装入一遍，虽然它们的内容其实并无改变。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

至于“局部段描述表”，则 Linux 内核并不使用局部段，所以将 LDTR 设成 0（见 264～265 行）。

至此，初始化的第一阶段已经完成。在 SMP 结构的系统中，主 CPU 和次 CPU 的代码又要分道扬镳了。我们再往下看。

==================== arch/i386/kernel/head.S 267 279 ====================

[startup_32()]

```
267  #ifdef CONFIG_SMP
268       movb ready, %cl
269       cmpb $1,%cl
270       je 1f              # the first CPU calls start_kernel
271                     # all other CPUs call initialize_secondary
272       call SYMBOL_NAME(initialize_secondary)
273       jmp L6
274  1:
275  #endif
276       call SYMBOL_NAME(start_kernel)
277  L6:
278       jmp L6             # main should never return here, but
279                     # just in case, we know what happens.
```

在 SMP 结构的系统中，用变量 ready 对已经执行了初始化第一阶段的 CPU 进行计数（见 248 行）。主 CPU 是首先执行初始化第一阶段的，所以如果此时 ready 为 1 就表明当前 CPU 是主 CPU，否则就是次 CPU。主 CPU 在完成了初始化第一阶段以后还任重道远，所以在 276 行调用 start_kernel()继续进行第一阶段的初始化。而次 CPU 则是“大树底下好乘凉”，主 CPU 已经为之作好了准备，因此直接就通过 initialize_secondary()转入其空转进程，有关的过程读者已经在前一章中看到过了。不过，次 CPU 的运行要到主 CPU 基本上完成了第二阶段的初始化时才会启动，而且主 CPU 在每启动一个次 CPU 以后都会等待其完成初始化。

==================== arch/i386/kernel/smpboot.c 474 486 ====================

[startup_32()>initialize_secondary()]

```
/*
 * Everything has been set up for the secondary
 * CPUs - they just need to reload everything
 * from the task structure
 * This function must not return.
 */
void __init initialize_secondary(void)
{
     /*
      * We don't actually need to load the full TSS,
      * basically just the stack pointer and the eip.
      */

     asm volatile(
          "movl %0,%%esp\n\t"
          "jmp *%1"
          :
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
            :"r" (current->thread.esp),"r" (current->thread.eip));
}
```

这里，次 CPU 的跳转地址和堆栈指针都是主 CPU 为之准备好的。

表面上看来，从 start_kernel()或 initialize_secondary()返回以后就落入了一个无限循环（277～278 行），但是读者将会看到对这两个函数的调用是不会返回的。对于主 CPU 或者单处理器系统中的 CPU，下面就是系统初始化的第二阶段了。

10.3 系统初始化（第二阶段）

从某种意义上说，函数 start_kernel()就好像一般可执行程序中的主函数 main()，系统在进入这个函数之前已经进行了一些最低限度的初始化，为这个函数的执行建立起了一个环境，创造了必要的条件。当然，这个函数还要继续进行内核的初始化，实际上甚至可以说内核的初始化这才真正开始。但是这种初始化与在此之前的初始化毕竟不同，是较高层次上的初始化。这也是为什么从这里开始的代码基本上是 C 代码，而在此之前则都是汇编代码的原因。这个函数的代码在 init/main.c 中：

==================== init/main.c 516 531 ====================

```
/*
 *   Activate the first processor.
 */

asmlinkage void __init start_kernel(void)
{
     char * command_line;
     unsigned long mempages;
     extern char saved_command_line[];
/*
 * Interrupts are still disabled. Do necessary setups, then
 * enable them
 */
     lock_kernel();
     printk(linux_banner);
     setup_arch(&command_line);
```

这里首先通过 printk()在屏幕上显示出内核的版本信息，这些信息是在编译时生成的，具体可参考 init/version.c 中的有关内容，此处不加赘述：

==================== init/version.c 24 26 ====================

```
const char *linux_banner =
     "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
     LINUX_COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

然后是 setup_arch()。顾名思义，这个函数所处理的是系统结构的设置，这就是初始化第二阶段的主体。不过我们在这里并不关心为一些特殊要求和情况所作的考虑，所以从函数中抽去一些条件编译的代码。例如，CONFIG_VISWS 是为 SUN 公司的工作站而设的，CONFIG_BLK_DEV_RAM 是为

PDF-XCHANGE Click to buy NOW! www.docu-track.com

RAMDISK 而设的，这些都不在我们关心之列，有兴趣或需要的读者可以自行阅读有关代码。函数 setup_arch()的代码在 arch/i386/kernel/setup.c，我们分段阅读。

==================== arch/i386/kernel/setup.c 598 604 ====================

[start_kernel()>setup_arch()]

```
void __init setup_arch(char **cmdline_p)
{
	unsigned long bootmap_size;
	unsigned long start_pfn, max_pfn, max_low_pfn;
	int i;

#ifdef CONFIG_VISWS
```

==================== arch/i386/kernel/setup.c 606 620 ====================

```
#endif

	ROOT_DEV = to_kdev_t(ORIG_ROOT_DEV);
	drive_info = DRIVE_INFO;
	screen_info = SCREEN_INFO;
	apm_info.bios = APM_BIOS_INFO;
	if( SYS_DESC_TABLE.length != 0 ) {
		MCA_bus = SYS_DESC_TABLE.table[3] &0x2;
		machine_id = SYS_DESC_TABLE.table[0];
		machine_submodel_id = SYS_DESC_TABLE.table[1];
		BIOS_revision = SYS_DESC_TABLE.table[2];
	}
	aux_device_present = AUX_DEVICE_INFO;

#ifdef CONFIG_BLK_DEV_RAM
	. . . . . .
```

==================== arch/i386/kernel/setup.c 624 626 ====================

```
#endif
	setup_memory_region();

```

代码中的 ROOT_DEV 是个全局量，显而易见是根设备的设备号，而 ORIG_ROOT_DEV 就是前述参数块中的一个 16 位设备号，代表着从中引导内核映象的设备，由引导辅助程序 setup 在引导时加以设置并传递给内核。这里先假定引导设备就是根设备，如果在引导命令行中另有指定，则在后面处理命令行时再加修正。其它的 drive_info、screen_info 等也与此类似，有关的信息均来自参数块，实际上是来自 BIOS。其中 MCA_bus 表示系统中是否配备了 PS/2 的 Micro Channel 总线，而 machine_id 和 machine_submodel_id 则显然是具体 PC 机的标号。

如前所述，BIOS 的功能并不只是引导操作系统内核，还担负着加电以后的自检和对资源的扫描探测，其中就包括了对物理内存的自检和扫描（想必读者在开机时看到过 BIOS 在此阶段中显示的信息），对于在这个阶段中获得的内存信息可以通过 BIOS 调用“ int 0x15”加以查询。由于在 Linux 内核中不能作 BIOS 调用，所以由 setup 在引导阶段代为查询，并根据获得的信息生成一幅物理内存构成图，称为 e820 图，再通过参数块传给内核，使内核能知道系统中内存资源的配置。之所以称为 e820 图，是因为在通过“ int 0x15”查询内存的构成时要把调用参数之一设置成 0xe820。这里的目的是把这个图复

PDF-XCHANGE Click to buy NOW! www.docu-track.com

制到一个安全的地方，因为参数块只是暂时存放在 empty_zero_page 中，这个页面最后要用于其它目的。

==================== arch/i386/kernel/setup.c 491 525 ====================

[start_kernel()>setup_arch()>setup_memory_region()]

```
/*
 * Do NOT EVER look at the BIOS memory size location.
 * It does not work on many machines.
 */
#define LOWMEMSIZE()   (0x9f000)

void __init setup_memory_region(void)
{
     char *who = "BIOS-e820";

     /*
      * Try to copy the BIOS-supplied E820-map.
      *
      * Otherwise fake a memory map; one section from 0k->640k,
      * the next section from 1mb->appropriate_mem_k
      */
     if (copy_e820_map(E820_MAP, E820_MAP_NR) < 0) {
          unsigned long mem_size;

          /* compare results from other methods and take the greater */
          if (ALT_MEM_K < EXT_MEM_K) {
               mem_size = EXT_MEM_K;
               who = "BIOS-88";
          } else {
               mem_size = ALT_MEM_K;
               who = "BIOS-e801";
          }

          e820.nr_map = 0;
          add_memory_region(0, LOWMEMSIZE(), E820_RAM);
          add_memory_region(HIGH_MEMORY, (mem_size << 10) - HIGH_MEMORY, E820_RAM);
     }
     printk("BIOS-provided physical RAM map:\n");
     print_memory_map(who);
} /* setup_memory_region */
```

为方便阅读，我们把参数块中与此有关的常数的定义列出于下（见 include/asm-i386/e820.h 和 arch/i386/kernel/setup.c）：

==================== include/asm-i386/e820.h 15 17 ====================

```
#define E820MAP    0x2d0         /* our map */
#define E820MAX    32       /* number of entries in E820MAP */
#define E820NR     0x1e8         /* # entries in E820MAP */
```

==================== arch/i386/kernel/setup.c 158 159 ====================

```
158 #define E820_MAP_NR (*(char*) (PARAM+E820NR))
159 #define E820_MAP    ((struct e820entry *) (PARAM+E820MAP))
```

其中，E820_MAP 是个 e820entry 数据结构指针，存放在参数块中位移为 0x2d0 的地方，这个数据结构的定义在 include/asm-i386/e820.h 中：

==================== include/asm-i386/e820.h 28 37 ====================

```
28  struct e820map {
29      int nr_map;
30      struct e820entry {
31       unsigned long long addr;    /* start of memory segment */
32       unsigned long long size;    /* size of memory segment */
33       unsigned long type;         /* type of memory segment */
34      } map[E820MAX];
35  };
36
37  extern struct e820map e820;
```

由此可见，所谓“map”是一个 e820entry 结构数组，数组中的每一项都是对一个物理内存区间的描述。代码中先通过 copy_e820_map()复制，如果发现数组中的信息可疑，就放弃复制而根据参数块中的其它信息生成（猜测）出有关的内容（508～521 行）。函数 copy_e820_map()的代码也在 arch/i386/kernel/setup.c 中：

==================== arch/i386/kernel/setup.c 440 489 ====================

[start_kernel()>setup_arch()>setup_memory_region()>copy_e820_map()]

```
440 /*
441  * Copy the BIOS e820 map into a safe place.
442  *
443  * Sanity-check it while we're at it..
444  *
445  * If we're lucky and live on a modern system, the setup code
446  * will have given us a memory map that we can use to properly
447  * set up memory.  If we aren't, we'll fake a memory map.
448  *
449  * We check to see that the memory map contains at least 2 elements
450  * before we'll use it, because the detection code in setup.S may
451  * not be perfect and most every PC known to man has two memory
452  * regions: one from 0 to 640k, and one from 1mb up.  (The IBM
453  * thinkpad 560x, for example, does not cooperate with the memory
454  * detection code.)
455  */
456 static int __init copy_e820_map(struct e820entry * biosmap, int nr_map)
457 {
458      /* Only one memory region (or negative)? Ignore it */
459      if (nr_map < 2)
460           return -1;
461
462      do {
463           unsigned long long start = biosmap->addr;
```

```
        unsigned long long size = biosmap->size;
        unsigned long long end = start + size;
        unsigned long type = biosmap->type;

        /* Overflow in 64 bits? Ignore the memory map. */
        if (start > end)
            return -1;

        /*
         * Some BIOSes claim RAM in the 640k - 1M region.
         * Not right. Fix it up.
         */
        if (type == E820_RAM) {
            if (start < 0x100000ULL && end > 0xA0000ULL) {
                if (start < 0xA0000ULL)
                    add_memory_region(start, 0xA0000ULL-start, type);
                if (end <= 0x100000ULL)
                    continue;
                start = 0x100000ULL;
                size = end - start;
            }
        }
        add_memory_region(start, size, type);
    } while (biosmap++,--nr_map);
    return 0;
}
```

如上所述，每个 e820entry 结构都是对一个物理内存区间的描述。从数据结构的定义中也可以看出，一个物理内存区间必须是同一类型的。如果有一片地址连续的物理内存空间，其一部分是 RAM，而另一部分是 ROM，那就要分成两个区间。即使同属 RAM，如果其中一部分要保留用于特殊目的，那也属于一个不同的分区。文件 include/asm-i386/e820.h 中定义了 4 种不同的类型：

==================== include/asm-i386/e820.h 19 22 ====================

```
#define E820_RAM    1
#define E820_RESERVED   2
#define E820_ACPI   3 /* usable as RAM once ACPI tables have been read */
#define E820_NVS    4
```

其中 E820_NVS 表示“ Non-Volatile Storage”，即“不挥发”存储器，包括 ROM、EPROM、Flash 存储器等等。

在 PC 机中，对于最初 1MB 存储空间的使用是特殊的。开头 640KB（即 0x0～0x9FFFF）为 RAM。从 0xA0000 开始的空间则用于 CGA、EGA、VGA 等图形卡。现在已经很少有人用 EGA 或 VGA（更不用提 CGA）了，但是不管是什么图形卡，开机时总是工作于 EGA 或 VGA 模式。从 0xF0000 开始到 0xFFFFF，即最高的 64KB，那就是在 EPROM 或 Flash 存储器中的 BIOS。所以，只要有 BIOS 存在，就至少也得有两个区间，如果 nr_map 小于 2 就一定错了。此外，如果一个区间的起点地址加上长度以后反而小了，那就说明发生了溢出，一个 64 位的地址（类型为 unsigned long long）发生溢出当然一定

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

是错了。由于 BIOS 和图形卡存储空间的存在，本来可以是连续、均质的 RAM 空间就不连续、不均匀了。当然，现在已经不会再有人会糊涂到有意地设计出这么一种存储空间结构了。之所以会有这样的结构，是因为在 PC 的早期人们觉得“个人计算机”有 640KB 的 RAM 空间已经是匪夷所思，而且当时 Intel X86 系列 CPU 的总的寻址能力也只有 1MB（当时有一种很流行的处理器 Z80，只有 64KB 的寻址能力）。技术的发展往往会使人“大跌眼镜”，使原来合理而且权威的设计变得可笑。后来，1MB 的边界很快就被冲破了，于是把 1MB 以上的空间称为“ HIGH_MEMORY”。这个称呼一直沿用到了现在，代码中的常数 HIGH_MEMORY 就定义为（1024 × 1024）。现在，配备了 128MB RAM 的 PC 机已经是很普通的了。但是，作为一种系统结构，在最初 1MB 的 RAM 空间中还得留下这么个空洞，否则便不能与业已存在的硬件和软件兼容。所以，代码中对于每个区间都调用 add_memory_region()，将其参数复制到数据结构 e820 内的数组中。而关键在于：当一个 RAM 区间的起点在 0xA0000 以下，而终点在 1MB 以上时，就要将这个区间拆开成两个区间，中间跳过从 0xA0000 到 1MB 边界之间的那一部分。不过，在特殊的情况下也可以通过引导命令行中的选择项改变这种空间结构。

回到 setup_arch()的代码中，继续往下看。

==================== arch/i386/kernel/setup.c 627 658 ====================

[start_kernel()>setup_arch()]

```
	if (!MOUNT_ROOT_RDONLY)
		root_mountflags &= ~MS_RDONLY;
	init_mm.start_code = (unsigned long) &_text;
	init_mm.end_code = (unsigned long) &_etext;
	init_mm.end_data = (unsigned long) &_edata;
	init_mm.brk = (unsigned long) &_end;

	code_resource.start = virt_to_bus(&_text);
	code_resource.end = virt_to_bus(&_etext)-1;
	data_resource.start = virt_to_bus(&_etext);
	data_resource.end = virt_to_bus(&_edata)-1;

	parse_mem_cmdline(cmdline_p);

#define PFN_UP(x)	(((x) + PAGE_SIZE-1) >> PAGE_SHIFT)
#define PFN_DOWN(x)	((x) >> PAGE_SHIFT)
#define PFN_PHYS(x)	((x) << PAGE_SHIFT)

/*
 * 128MB for vmalloc and initrd
 */
#define VMALLOC_RESERVE	(unsigned long)(128 << 20)
#define MAXMEM		(unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE)
#define MAXMEM_PFN	PFN_DOWN(MAXMEM)
#define MAX_NONPAE_PFN	(1 << 20)

	/*
	 * partially used pages are not usable - thus
	 * we are rounding upwards:
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
     */
    start_pfn = PFN_UP(__pa(&_end));

```

除了对数据结构 init_mm 和 code_resource 的设置以外，这里主要是对 parse_mem_cmdline()的调用。数据结构 init_mm 是系统中第一个进程 swapper 的存储空间控制结构，也是整个内核的 mm_struct 数据结构。这个数据结构代表着系统空间，不管是什么进程，只要一进入内核就进入了系统空间，就受这个数据结构控制。

在特殊的情况下，有的系统可能有特殊的 RAM 空间结构，此时可以通过引导命令行中的选择项改变存储空间的逻辑结构，使其正确地反映内存的物理结构。函数 parse_mem_cmdline()的作用就是分析命令行中的选择项，并据此对数据结构 e820 中的内容作出修正。这个函数的代码在 arch/i386/kernel/setup.c 中，但是我们不深入进去了，只是把代码中的一段注释抄录于下，让读者对这些选择项有个大致的印象：

==================== arch/i386/kernel/setup.c 539 545 ====================

```
        /*
         * "mem=nopentium" disables the 4MB page tables.
         * "mem=XXX[kKmM]" defines a memory region from HIGH_MEM
         * to <mem>, overriding the bios size.
         * "mem=XXX[KkmM]@XXX[KkmM]" defines a memory region from
         * <start> to <start>+<mem>, overriding the bios size.
         */
```

其中的第一项用于 CPU 为 Pentium，并且内核在编译时采用了 36 位地址（PAE 模式），又采用了 Pentium 的 PSE 功能，即采用 4MB 页面，但是在引导时却临时决定要采用 4KB 页面的情景。我们在本书中只关心 32 位地址、页面大小为 4KB 的模式，有兴趣或有需要的读者自可对 PAE 和 PSE 加以研究。

然后，代码中就地定义了一些宏操作和常数，这些定义都是下面要用到的。首先就是用在对 start_pfn 的计算（pfn 大概是“ Page Frame Number”的缩写），这个变量的值是个页面号，代表着内存中内核映象以上第一个可以动态分配的页面。内核映象的终点是_end，这是由 gcc 在编译和连接时自动生成的，从它的地址往上就是可以动态分配的空间了，宏操作 PFN_UP()根据这个地址计算出它上面的第一个页面边界。

继续看 setup_arch()的代码。

==================== arch/i386/kernel/setup.c 659 708 ====================

[start_kernel()>setup_arch()]

```
    /*
     * Find the highest page frame number we have available
     */
    max_pfn = 0;
    for (i = 0; i < e820.nr_map; i++) {
        unsigned long start, end;
        /* RAM? */
        if (e820.map[i].type != E820_RAM)
            continue;
        start = PFN_UP(e820.map[i].addr);
```

```
669             end = PFN_DOWN(e820.map[i].addr + e820.map[i].size);
670             if (start >= end)
671                 continue;
672             if (end > max_pfn)
673                 max_pfn = end;
674         }
675
676         /*
677          * Determine low and high memory ranges:
678          */
679         max_low_pfn = max_pfn;
680         if (max_low_pfn > MAXMEM_PFN) {
681             max_low_pfn = MAXMEM_PFN;
682     #ifndef CONFIG_HIGHMEM
683             /* Maximum memory usable is what is directly addressable */
684             printk(KERN_WARNING "Warning only %ldMB will be used.\n",
685                                     MAXMEM>>20);
686             if (max_pfn > MAX_NONPAE_PFN)
687                 printk(KERN_WARNING "Use a PAE enabled kernel.\n");
688             else
689                 printk(KERN_WARNING "Use a HIGHMEM enabled kernel.\n");
690     #else /* !CONFIG_HIGHMEM */
691     #ifndef CONFIG_X86_PAE
692             if (max_pfn > MAX_NONPAE_PFN) {
693                 max_pfn = MAX_NONPAE_PFN;
694                 printk(KERN_WARNING "Warning only 4GB will be used.\n");
695                 printk(KERN_WARNING "Use a PAE enabled kernel.\n");
696             }
697     #endif /* !CONFIG_X86_PAE */
698     #endif /* !CONFIG_HIGHMEM */
699         }
700
701     #ifdef CONFIG_HIGHMEM
702         highstart_pfn = highend_pfn = max_pfn;
703         if (max_pfn > MAXMEM_PFN) {
704             highstart_pfn = MAXMEM_PFN;
705             printk(KERN_NOTICE "%ldMB HIGHMEM available.\n",
706                 pages_to_mb(highend_pfn - highstart_pfn));
707         }
708     #endif
```

在数据结构 e820 中积累起各个物理内存区间的信息以后，要从这些信息中归纳出一项重要的数据，那就是 RAM 空间的顶点 max_pfn。代码中通过一个 for 循环扫描 e820 中的数组，通过比较和计算得出此项数据。宏操作 PFN_UP()和 PFN_DOWN()的定义见上面的 641 和 642 行，作用是将内存地址转换成页面号，因此 max_pfn 所代表的是系统中最高的 RAM 页面。不过，实际可以使用多大的物理内存不光取决于物理上的配备以及整个地址空间的大小，还进一步受到内核的虚存空间，即所谓系统空间大小的限制。读者在第 2 章中曾看到，Linux 的内核将整个 32 位地址空间分成两部分，其中 0xC0000000

以上的 1GB 为系统空间，从系统空间到物理内存的映射基本上是线性映射。如果物理内存的大小超过了 1GB，就不能从内核中访问到整个物理内存。另一方面，虽然从系统空间到物理内存的映射基本上是线性映射、因而不会将同一物理内存页面重复地映射到系统空间中，但也还是有例外的。我们在第 2 章中提到过 vmalloc()，它就是在原有的线性映射以外另行分配一块系统空间（虚拟地址），并建立起与物理页面的映射。此外，如果采用 RAMDISK，则也有类似的情况。这样，由于同一物页面有可能耗用一个以上的虚存页面，可以在系统空间中直接（不需要通过临时改变映射）访问的物理页面数量就又减少了，不过这一部分虚存空间的大小限于 128MB。因此，从可以被内核直接访问的角度，理论上最大的 RAM 空间容量为 1024MB - 128MB = 896MB，这就是常数 MAXMEM，相应的页面号为 MAXMEM_PFN。这个数值有可能小于 max_pfn，所以另有一个变量 max_low_pfn 用来记录这个数值。下面是几个有关常数的定义，分别见于 arch/i386/kernel/setup.c 和 include/asm-i386/page.h：

==================== arch/i386/kernel/setup.c 645 650 ====================

```
645 /*
646  * 128MB for vmalloc and initrd
647  */
648 #define VMALLOC_RESERVE (unsigned long)(128 << 20)
649 #define MAXMEM          (unsigned long)(-PAGE_OFFSET-VMALLOC_RESERVE)
650 #define MAXMEM_PFN PFN_DOWN(MAXMEM)
```

==================== include/asm-i386/page.h 81 81 ====================

```
81  #define __PAGE_OFFSET       (0xC0000000)
```

所以，对于 Linux 内核，如果系统中配备了 896MB 以上的 RAM，就需要超过 32 位的寻址能力，此时要选用编译选择项 CONFIG_HIGHMEM，注意这里所谓 HIGHMEM 是指 4GB 虚存空间，而不是前面讲的 1MB，不要搞混淆了。从代码中的 682～698 行可以看出，如果配备了 896MB 以上的 RAM，而又不选用 CONFIG_HIGHMEM，则只能使用其中的 896MB。如果选用了 CONFIG_HIGHMEM，则又要看是否选用 Pentium 的 PAE 模式，那些我们就不关心了，反正现在大概还没有多少人要用 896MB 以上的物理内存（几年以后也许又要大跌眼镜）。还要指出，这只是对 32 位结构的 i386 而言，对新的 ia64 结构当然要另作别论。

我们再往下看。

==================== arch/i386/kernel/setup.c 709 779 ====================

[start_kernel()>setup_arch()]

```
709     /*
710      * Initialize the boot-time allocator (with low memory only):
711      */
712     bootmap_size = init_bootmem(start_pfn, max_low_pfn);
713 
714     /*
715      * Register fully available low RAM pages with the bootmem allocator.
716      */
717     for (i = 0; i < e820.nr_map; i++) {
718          unsigned long curr_pfn, last_pfn, size;
719          /*
720           * Reserve usable low memory
```

```
		 */
		if (e820.map[i].type != E820_RAM)
			continue;
		/*
		 * We are rounding up the start address of usable memory:
		 */
		curr_pfn = PFN_UP(e820.map[i].addr);
		if (curr_pfn >= max_low_pfn)
			continue;
		/*
		 * ... and at the end of the usable range downwards:
		 */
		last_pfn = PFN_DOWN(e820.map[i].addr + e820.map[i].size);

		if (last_pfn > max_low_pfn)
			last_pfn = max_low_pfn;

		/*
		 * .. finally, did all the rounding and playing
		 * around just make the area go away?
		 */
		if (last_pfn <= curr_pfn)
			continue;

		size = last_pfn - curr_pfn;
		free_bootmem(PFN_PHYS(curr_pfn), PFN_PHYS(size));
	}
	/*
	 * Reserve the bootmem bitmap itself as well. We do this in two
	 * steps (first step was init_bootmem()) because this catches
	 * the (very unlikely) case of us accidentally initializing the
	 * bootmem allocator with an invalid RAM area.
	 */
	reserve_bootmem(HIGH_MEMORY, (PFN_PHYS(start_pfn) +
			 bootmap_size + PAGE_SIZE-1) - (HIGH_MEMORY));

	/*
	 * reserve physical page 0 - it's a special BIOS page on many boxes,
	 * enabling clean reboots, SMP operation, laptop functions.
	 */
	reserve_bootmem(0, PAGE_SIZE);

#ifdef CONFIG_SMP
	/*
	 * But first pinch a few for the stack/trampoline stuff
	 * FIXME: Don't need the extra page at 4K, but need to fix
	 * trampoline before removing it. (see the GDT stuff)
	 */
	reserve_bootmem(PAGE_SIZE, PAGE_SIZE);
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
770      smp_alloc_memory(); /* AP processor realmode stacks in low memory*/
771  #endif
772
773  #ifdef CONFIG_X86_IO_APIC
774      /*
775       * Find and reserve possible boot-time SMP configuration:
776       */
777      find_smp_config();
778  #endif
779      paging_init();
```

首先通过 init_bootmem()为物理内存页面管理机制的建立作些准备，为整个物理内存建立起一个页面位图。这个位图建立在从 start_pfn 开始的地方。也就是说，把内核映象终点_end 上方的若干页面用作物理页面位图。在此之前的代码中已经搞清楚了物理内存顶点所在的页面号是 max_low_pfn，所以整个物理内存的页面号一定在 0 至 max_low_pfn 这个范围内。可是，在这个范围中可能有空洞，另一方面也并不是所有的物理内存页面都可以动态分配。建立这个位图的目的就是要搞清楚哪一些物理内存页面是可以动态分配的。

==================== mm/bootmem.c 283 288 ====================

[start_kernel()>setup_arch()>init_bootmem()]

```
283  unsigned long __init init_bootmem (unsigned long start, unsigned long pages)
284  {
285      max_low_pfn = pages;
286      min_low_pfn = start;
287      return(init_bootmem_core(&contig_page_data, start, 0, pages));
288  }
```

操作的主体是 init_bootmem_core()，这个函数对 pg_data_t 数据结构 contig_page_data 进行初始化，读者已经在第 2 章的“几个重要的数据结构和函数”一节中看到过这种数据结构的定义。每个 pg_data_t 数据结构都代表着一片均匀的、连续的内存空间，称为一个“节点”。在连续空间 UMA 结构中只有一个节点 contig_page_data，而在 NUMA 结构或不连续空间 UMA 结构中则有多个这样的数据结构。系统中各个节点的 pg_data_t 数据结构通过指针 node_next 连接在一起成为一个链，全局量 pgdat_list 则指向这个链。显然，contig_page_data 是链中的第一个节点。这里先假定整个物理内存空间为均匀的、连续的，以后若发现这个假定不能成立则再加以修正，再将新的 pg_data_t 结构加入到链中。数据结构 contig_page_data 的初始内容为：

==================== mm/numa.c 14 15 ====================

```
14   static bootmem_data_t contig_bootmem_data;
15   pg_data_t contig_page_data = { bdata: &contig_bootmem_data };
```

在 pg_data_t 结构中有个指针 bdata，指向一个 bootmem_data_t 数据结构，其类型定义如下（include/linux/bootmem.h）：

==================== include/linux/bootmem.h 20 30 ====================

```
20   /*
21    * node_bootmem_map is a map pointer - the bits represent all physical
```

```
 * memory pages (including holes) on the node.
 */
typedef struct bootmem_data {
     unsigned long node_boot_start;
     unsigned long node_low_pfn;
     void *node_bootmem_map;
     unsigned long last_offset;
     unsigned long last_pos;
} bootmem_data_t;
```

结构中的 node_boot_start 表示系统引导以后存在的第一个物理内存页面(而与引导过程本身无关)，从调用 init_bootmem_core()时的参数 start 可以看出是 0；node_low_pfn 则表示物理内存的顶点，最高不超过 896MB。结构中的指针 node_bootmem_map 指向一个“保留页面位图”，位图中的每一位都代表着物理内存中一个需要保留，或者不存在，从而不能用于动态分配的页面。函数 init_bootmem_core()的代码在 mm/bootmem.c 中：

==================== mm/bootmem.c 41 65 ====================

[start_kernel()>setup_arch()>init_bootmem()>init_bootmem_core()]

```
/*
 * Called once to set up the allocator itself.
 */
static unsigned long __init init_bootmem_core (pg_data_t *pgdat,
     unsigned long mapstart, unsigned long start, unsigned long end)
{
     bootmem_data_t *bdata = pgdat->bdata;
     unsigned long mapsize = ((end - start)+7)/8;

     pgdat->node_next = pgdat_list;
     pgdat_list = pgdat;

     mapsize = (mapsize + (sizeof(long) - 1UL)) & ~(sizeof(long) - 1UL);
     bdata->node_bootmem_map = phys_to_virt(mapstart << PAGE_SHIFT);
     bdata->node_boot_start = (start << PAGE_SHIFT);
     bdata->node_low_pfn = end;

     /*
      * Initially all pages are reserved - setup_arch() has to
      * register free RAM areas explicitly.
      */
     memset(bdata->node_bootmem_map, 0xff, mapsize);

     return mapsize;
}
```

注意这里参数 start 的值为 0，而 mapstart 的值才是上一层函数 init_bootmem()中的 start，即内核映象以上第一个页面的起点，参数 end 的值则为物理内存的顶点 max_low_pfn。

我们基本上把这个函数以及 setup_arch()中的 717～761 行留给读者自己结合着阅读。这里只作简略

的说明：

函数 free_bootmem()对一个 contig_page_data 结构中的位图进行操作，将其中的某些位清 0，表示相应的物理内存页面可以投入分配。而 reserve_bootmem()则正好相反，一开始时把位图中的所有位都设置成 1，假定全部都不能用于动态分配，然后根据 e820 数据结构中的内容以及一些特殊的区间和页面加以找补。其中特别加以保留的有：

- 从 HIGH_MEMORY，即 1MB 边界开始，直到(start_pfn + bootmap_size)所在的页面为止。这些是内核映象和“保留页面位图”本身所在的页面。
- 页面 0，即起始地址为 0 的页面。BIOS 通常用这个页面保存一些与引导以及 BIOS 本身有关的信息，所以也要加以保留。
- 对于 SMP 结构的系统，页面 1，即起始地址为 PAGE_SIZE 的页面也要保留，次 CPU 转入运行时需要用这个页面作为“跳板”（见第 9 章）。
- 此外，还有一些特殊用途的页面，例如用作 RAMDISK 的页面。不过这些页面不是在这儿保留的。

对于采用 CONFIG_X86_IO_APIC 的 SMP 系统，还要通过 find_smp_config()寻找由 BIOS 和引导装入程序设置在基本内存（640KB）中的多处理器配置表，不过我们在这里不深入到这个函数中去了。

前面已经建立了为内存页面管理所需的数据结构，现在是进一步完善页面映射机制，并且建立起内存页面管理机制的时候了（779 行）。函数 paging_init()的代码在 arch/i386/mm/init.c 中：

==================== arch/i386/mm/init.c 437 484 ====================

[start_kernel()>setup_arch()>paging_init()]

```
/*
 * paging_init() sets up the page tables - note that the first 8MB are
 * already mapped by head.S.
 *
 * This routines also unmaps the page at virtual kernel address 0, so
 * that we can trap those pesky NULL-reference errors in the kernel.
 */
void __init paging_init(void)
{
     pagetable_init();

     __asm__( "movl %%ecx,%%cr3\n" ::"c"(__pa(swapper_pg_dir)));

#if CONFIG_X86_PAE
     /*
      * We will bail out later - printk doesnt work right now so
      * the user would just see a hanging kernel.
      */
     if (cpu_has_pae)
          set_in_cr4(X86_CR4_PAE);
#endif

     __flush_tlb_all();

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#ifdef CONFIG_HIGHMEM
     kmap_init();
#endif
     {
          unsigned long zones_size[MAX_NR_ZONES] = {0, 0, 0};
          unsigned int max_dma, high, low;

          max_dma = virt_to_phys((char *)MAX_DMA_ADDRESS) >> PAGE_SHIFT;
          low = max_low_pfn;
          high = highend_pfn;

          if (low < max_dma)
               zones_size[ZONE_DMA] = low;
          else {
               zones_size[ZONE_DMA] = max_dma;
               zones_size[ZONE_NORMAL] = low - max_dma;
#ifdef CONFIG_HIGHMEM
               zones_size[ZONE_HIGHMEM] = high - low;
#endif
          }
          free_area_init(zones_size);
     }
     return;
}
```

首先通过 pagetabte_init()扩充由 startup_32()在第一阶段中创建的页面映射目录和页面映射表（见 arch/i386/kernel/head.S 中的 389 行和 98～102 行）。当初因为不知道内存到底有多大，所以只为开始的 8MB 建立了映射。现在既然已经有了关于内存的详细信息，就可以根据这些信息加以扩充和修改，建立完整的从系统空间到整个物理存储空间的线性映射了。函数 pagetable_init()的代码也在同一文件中：

==================== arch/i386/mm/init.c 314 414 ====================

[start_kernel()>setup_arch()>paging_init()>pagetable_init()]

```
static void __init pagetable_init (void)
{
     unsigned long vaddr, end;
     pgd_t *pgd, *pgd_base;
     int i, j, k;
     pmd_t *pmd;
     pte_t *pte;

     /*
      * This can be zero as well - no problem, in that case we exit
      * the loops anyway due to the PTRS_PER_* conditions.
      */
     end = (unsigned long)__va(max_low_pfn*PAGE_SIZE);

     pgd_base = swapper_pg_dir;
#if CONFIG_X86_PAE
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    for (i = 0; i < PTRS_PER_PGD; i++) {
        pgd = pgd_base + i;
        __pgd_clear(pgd);
    }
#endif
    i = __pgd_offset(PAGE_OFFSET);
    pgd = pgd_base + i;

    for (; i < PTRS_PER_PGD; pgd++, i++) {
        vaddr = i*PGDIR_SIZE;
        if (end && (vaddr >= end))
            break;
#if CONFIG_X86_PAE
        pmd = (pmd_t *) alloc_bootmem_low_pages(PAGE_SIZE);
        set_pgd(pgd, __pgd(__pa(pmd) + 0x1));
#else
        pmd = (pmd_t *)pgd;
#endif
        if (pmd != pmd_offset(pgd, 0))
            BUG();
        for (j = 0; j < PTRS_PER_PMD; pmd++, j++) {
            vaddr = i*PGDIR_SIZE + j*PMD_SIZE;
            if (end && (vaddr >= end))
                break;
            if (cpu_has_pse) {
                unsigned long __pe;

                set_in_cr4(X86_CR4_PSE);
                boot_cpu_data.wp_works_ok = 1;
                __pe = _KERNPG_TABLE + _PAGE_PSE + __pa(vaddr);
                /* Make it "global" too if supported */
                if (cpu_has_pge) {
                    set_in_cr4(X86_CR4_PGE);
                    __pe += _PAGE_GLOBAL;
                }
                set_pmd(pmd, __pmd(__pe));
                continue;
            }

            pte = (pte_t *) alloc_bootmem_low_pages(PAGE_SIZE);
            set_pmd(pmd, __pmd(_KERNPG_TABLE + __pa(pte)));

            if (pte != pte_offset(pmd, 0))
                BUG();

            for (k = 0; k < PTRS_PER_PTE; pte++, k++) {
                vaddr = i*PGDIR_SIZE + j*PMD_SIZE + k*PAGE_SIZE;
                if (end && (vaddr >= end))
                    break;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
                        *pte = mk_pte_phys(__pa(vaddr), PAGE_KERNEL);
                    }
                }
        }

        /*
         * Fixed mappings, only the page table structure has to be
         * created - mappings will be set by set_fixmap():
         */
        vaddr = __fix_to_virt(__end_of_fixed_addresses - 1) & PMD_MASK;
        fixrange_init(vaddr, 0, pgd_base);

#if CONFIG_HIGHMEM
        /*
         * Permanent kmaps:
         */
        vaddr = PKMAP_BASE;
        fixrange_init(vaddr, vaddr + PAGE_SIZE*LAST_PKMAP, pgd_base);

        pgd = swapper_pg_dir + __pgd_offset(vaddr);
        pmd = pmd_offset(pgd, vaddr);
        pte = pte_offset(pmd, vaddr);
        pkmap_page_table = pte;
#endif

#if CONFIG_X86_PAE
        /*
         * Add low memory identity-mappings - SMP needs it when
         * starting up on an AP from real-mode. In the non-PAE
         * case we already have these mappings through head.S.
         * All user-space mappings are explicitly cleared after
         * SMP startup.
         */
        pgd_base[0] = pgd_base[USER_PTRS_PER_PGD];
#endif
}
```

我们把这段代码留给读者作为对第 2 章中有关内容的复习。注意，这里对页面映射目录 swapper_pg_dir 中目录项的设置是从下标__pgd_offset(PAGE_OFFSET)开始，即从虚拟地址 0xC0000000 中的最高 10 位 0x300 开始。页面映射目录的大小为一个页面，实际上就是个大小为 1024 的指针数组，这里从下标 0x300，即 768 开始，这是为系统空间准备的。一旦内核线程进入正常运行以后，就不会使用 0xC0000000 以下的虚拟地址了，所以这个映射目录中前 3/4 的目录项最终都要设置成 0，不过现在还不到时候。

设置好页面映射目录以后，448 行的汇编指令 movl 将目录的起始地址 swapper_pg_dir 送入控制寄存器%cr3。读者也许还记得，前面在 startup_32()中已经把这个地址设置进%cr3 中了（99 行），为什么这里还要再设置一次呢？这是因为每当设置%cr3 时 CPU 就会将页面映射目录所在的页面装入 CPU 内

PDF-XCHANGE Click to buy NOW! www.docu-track.com

部高速缓存中的 TLB 部分。现在内存中（实际上是高速缓存中，见下）的映射目录变了，就要再让 CPU 装入一次。由于页面映射机制本来就是开启着的，所以从这条指令以后就扩大了系统空间中有映射区间的大小，使映射的范围覆盖到整个物理内存（HIGHMEM 除外）。不过，这里所谓“内存中”是逻辑意义上的，实际上此时 swapper_pg_dir 中已经改变的目录项很可能还在高速缓存中，所以还要通过__flush_tlb_all()将高速缓存中的内容冲刷到内存中，这样才能保证内存中映射目录内容的一致性。

如前所述，要是系统中配备了高于 896MB 以上的内存，而 CPU 又只有 32 位的寻址能力，那就得选用 CONFIG_HIGHMEM 选项。那么，选用了 CONFIG_HIGHMEM 选项以后又是怎样实现对 896MB 以上物理页面的映射呢？根据所谓“抽屉原理”，如果要将多于 n 个的物件（在这里是物理页面）放入 n 个抽屉（在这里是属于系统空间的页面映射表项），则至少有一个抽屉里要存放多于一个的物件。但是，一个页面表项在同一时间内又确实只能映射一个物理页面，那就说明至少有一个页面表项要在不同的时间里映射到不同的物理页面。对于选用了 CONFIG_HIGHMEM 的系统，内核中设置了一个全局的 pte_t 指针 kmap_pte，指向页面映射表中的一个表项，这个表项将动态地映射到不同的物理页面。每当要访问一个属于“高内存”的物理页面时，就要先改变这个表项。函数 kmap_init()的作用主要就是设置好指针 kmap_pte。

在第 2 章中，读者曾看到在代表着存储节点的 pg_data_t 数据结构中的一个数组 node_zones[]，其大小为 3。通过这个数组，内核将每个节点中的物理页面划分成三个“管理区”（zone），即 ZONE_DMA、ZONE_NORMAL 以及 ZONE_HIGHMEM，其中 ZONE_HIGHMEM 只有在选用了 CONFIG_HIGHMEM 选项时才有效。为什么要这样划分呢？将 ZONE_HIGHMEM 区分出来是不言而喻的，因为访问这些“高内存”页面时要遵循特殊的步骤。进一步将“低内存”页面也分成两部分，则是因为 DMA 操作对目标页面的特殊要求。在 PC 机中，能够对之进行 DMA 操作的页面必须低于 0x1000000，即 16MB，相应的系统空间虚拟地址为 MAX_DMA_ADDRESS，这个常数在 include/asm-i386/dma.h 中定义为：

==================== include/asm-i386/dma.h 75 76 ====================

```
/* The maximum address that we can perform a DMA transfer to on this platform */
#define MAX_DMA_ADDRESS      (PAGE_OFFSET+0x1000000)
```

所以，把 16MB 以下的页面划入 ZONE_DMA 管理区，而从 16MB 以上直至 max_low_pfn 则划入 ZONE_NORMAL 管理区。代码中的 468～479 行准备下一个整数数组 zones_size[]，数组中记录了对物理内存的划分，然后将其传递给函数 free_area_init()，由这个函数根据数组 zones_size[]提供的指示在三个管理区中创建起空闲页面块队列。函数 free_area_init()及有关的代码在 mm/page_alloc.c 中，我们也把它留给读者。

至此，paging_init()的操作已经完成，我们回到 setup_arch()的代码中：

==================== arch/i386/kernel/setup.c 780 791 ====================

[start_kernel()>setup_arch()]

```
#ifdef CONFIG_X86_IO_APIC
     /*
      * get boot-time SMP configuration:
      */
     if (smp_found_config)
          get_smp_config();
#endif
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
#ifdef CONFIG_X86_LOCAL_APIC
        init_apic_mappings();
#endif

#ifdef CONFIG_BLK_DEV_INITRD
. . . . . .
```

==================== arch/i386/kernel/setup.c 807 852 ====================

```
#endif

        /*
         * Request address space for all standard RAM and ROM resources
         * and also for regions reported as reserved by the e820.
         */
        probe_roms();
        for (i = 0; i < e820.nr_map; i++) {
                struct resource *res;
                if (e820.map[i].addr + e820.map[i].size > 0x100000000ULL)
                        continue;
                res = alloc_bootmem_low(sizeof(struct resource));
                switch (e820.map[i].type) {
                case E820_RAM:     res->name = "System RAM"; break;
                case E820_ACPI:    res->name = "ACPI Tables"; break;
                case E820_NVS:     res->name = "ACPI Non-volatile Storage"; break;
                default:  res->name = "reserved";
                }
                res->start = e820.map[i].addr;
                res->end = res->start + e820.map[i].size - 1;
                res->flags = IORESOURCE_MEM | IORESOURCE_BUSY;
                request_resource(&iomem_resource, res);
                if (e820.map[i].type == E820_RAM) {
                        /*
                         *  We dont't know which RAM region contains kernel data,
                         *  so we try it repeatedly and let the resource manager
                         *  test it.
                         */
                        request_resource(res, &code_resource);
                        request_resource(res, &data_resource);
                }
        }
        request_resource(&iomem_resource, &vram_resource);

        /* request I/O space for devices used on all i[345]86 PCs */
        for (i = 0; i < STANDARD_IO_RESOURCES; i++)
                request_resource(&ioport_resource, standard_io_resources+i);

#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
        conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
849       conswitchp = &dummy_con;
850  #endif
851  #endif
852  }
```

物理内存页面当然是重要的资源，对这些资源的管理机制已经建立了。但是，从另一个角度看，地址空间本身，或者物理存储器在地址空间中的位置，也是一种资源，也要加以管理。为此，include/linux/ioport.h 中定义了一种 resource 数据结构：

==================== arch/i386/kernel/setup.c 807 852 ====================

```
11  /*
12   * Resources are tree-like, allowing
13   * nesting etc..
14   */
15  struct resource {
16       const char *name;
17       unsigned long start, end;
18       unsigned long flags;
19       struct resource *parent, *sibling, *child;
20  };
```

这种数据结构描述的是一片可以通过访问内存操作或 I/O 操作加以访问的连续空间，或者说操作对象在相应空间中的位置。这样的对象有 RAM、ROM 以及一些用于外部设备的器件。这里所关心的只是由这些器件所形成的逻辑意义上的连续存储空间或 I/O 地址空间，而并不关心每个这样的空间在物理上包含了几个芯片。另一方面，我们以前讲过，i386 处理器有独立于内存访问指令的 I/O 指令和独立于内存地址空间的 I/O 地址空间。虽然有些处理器没有独立的 I/O 指令和 I/O 地址空间，但是逻辑上还是能根据用途区分某项资源是属于存储性质的还是 I/O 性质的。每个 resource 结构通过其 parent、sibling、child 等指针互相连接在一起，形成一种树状结构。如果一个 resource 结构是另一个 resource 结构的子节点，则它所描述的空间是父节点所描述空间的一部分，或者是对父节点的具体化。内核中有两棵这样的树，一棵是 iomem_resource，另一棵是 ioport_resource，分别代表着两类不同性质的地址资源。两棵树的根本身也都是 resource 数据结构，不过这两个数据结构所描述的并不是用于具体操作对象的地址资源，而是概念上的整个地址空间（kernel/resource.c）。

===================== kernel/resource.c 18 19 =====================

```
18  struct resource ioport_resource = { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
19  struct resource iomem_resource = { "PCI mem", 0x00000000, 0xffffffff, IORESOURCE_MEM };
```

内核中设置了一个 resource 结构数组 rom_resources[]，用来记录系统中每个 ROM 空间的位置（arch/i386/kernel/setup.c）。所谓 ROM 实际上是指所有的不挥发只读存储器，包括 PROM、EPROM、Flash 存储器等存储器件。不过，这个数组中只包括 PC 机所固有的那些 ROM 空间，原则上都在母板上或第一块图形卡上，而不包括由其它接口卡提供的不挥发存储空间。以前说过，只要是 PC 机就至少有一个 ROM 空间，因为其 BIOS 总是在一个 ROM 空间中，并且所在的位置总是固定的。所以，数组中的第一个元素固定地初始化成“ System ROM”，位置是从 0xF0000 到 0xFFFFF。此外，图形接口中也有一个 ROM 空间，其位置应该是从 0xC0000 到 0xC7fff，不过这是要加以验证的。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== arch/i386/kernel/setup.c 325 330 ====================

```
/* System ROM resources */
#define MAXROMS 6
static struct resource rom_resources[MAXROMS] = {
     { "System ROM", 0xF0000, 0xFFFFF, IORESOURCE_BUSY },
     { "Video ROM", 0xc0000, 0xc7fff, IORESOURCE_BUSY }
};
```

除 BIOS 所在的 ROM 空间以外，其它的 ROM 空间都要通过扫捕来寻找或验证。不过，对 ROM 空间在内存中的位置是有限制的，它们只能出现在几个特定的区间，所以只需要扫描这几个特定的区间就可以了，函数 probe_roms()的代码在 arch/i386/kernel/setup.c 中：

==================== arch/i386/kernel/setup.c 332 395 ====================

[start_kernel()>setup_arch()>probe_roms()]

```
#define romsignature(x) (*(unsigned short *)(x) == 0xaa55)

static void __init probe_roms(void)
{
     int roms = 1;
     unsigned long base;
     unsigned char *romstart;

     request_resource(&iomem_resource, rom_resources+0);

     /* Video ROM is standard at C000:0000 - C7FF:0000, check signature */
     for (base = 0xC0000; base < 0xE0000; base += 2048) {
          romstart = bus_to_virt(base);
          if (!romsignature(romstart))
               continue;
          request_resource(&iomem_resource, rom_resources + roms);
          roms++;
          break;
     }

     /* Extension roms at C800:0000 - DFFF:0000 */
     for (base = 0xC8000; base < 0xE0000; base += 2048) {
          unsigned long length;

          romstart = bus_to_virt(base);
          if (!romsignature(romstart))
               continue;
          length = romstart[2] * 512;
          if (length) {
               unsigned int i;
               unsigned char chksum;

               chksum = 0;
               for (i = 0; i < length; i++)
                    chksum += romstart[i];
```

```

            /* Good checksum? */
            if (!chksum) {
                rom_resources[roms].start = base;
                rom_resources[roms].end = base + length - 1;
                rom_resources[roms].name = "Extension ROM";
                rom_resources[roms].flags = IORESOURCE_BUSY;

                request_resource(&iomem_resource, rom_resources + roms);
                roms++;
                if (roms >= MAXROMS)
                    return;
            }
        }
    }

    /* Final check for motherboard extension rom at E000:0000 */
    base = 0xE0000;
    romstart = bus_to_virt(base);

    if (romsignature(romstart)) {
        rom_resources[roms].start = base;
        rom_resources[roms].end = base + 65535;
        rom_resources[roms].name = "Extension ROM";
        rom_resources[roms].flags = IORESOURCE_BUSY;

        request_resource(&iomem_resource, rom_resources + roms);
    }
}
```

第一项 ROM 资源，即 BIOS 所在的 ROM 空间是固有的，所以不经扫描就直接通过 request_resource() 把它链入 iomem_resource 树中。有关的代码在 kernel/resource.c 中：

==================== kernel/resource.c 114 122 ====================
[start_kernel()>setup_arch()>probe_roms()>request_resource()]

```
int request_resource(struct resource *root, struct resource *new)
{
    struct resource *conflict;

    write_lock(&resource_lock);
    conflict = __request_resource(root, new);
    write_unlock(&resource_lock);
    return conflict ? -EBUSY : 0;
}
```

操作的主体是__request_resource()，其代码也在 kernel/resource.c 中：

==================== kernel/resource.c 66 93 ====================
[start_kernel()>setup_arch()>probe_roms()>request_resource()__request_resource()]

```
/* Return the conflict entry if you can't request it */
static struct resource * __request_resource(struct resource *root, struct resource *new)
{
        unsigned long start = new->start;
        unsigned long end = new->end;
        struct resource *tmp, **p;

        if (end < start)
                return root;
        if (start < root->start)
                return root;
        if (end > root->end)
                return root;
        p = &root->child;
        for (;;) {
                tmp = *p;
                if (!tmp || tmp->start > end) {
                        new->sibling = tmp;
                        *p = new;
                        new->parent = root;
                        return NULL;
                }
                p = &tmp->sibling;
                if (tmp->end < start)
                        continue;
                return tmp;
        }
}
```

由于 ROM 空间的位置限制在几个特定的区间，不需考虑与其它资源冲突的可能性，所以在probe_roms()中并不检查 request_resource()的返回值。除 BIOS 所在的 ROM 空间之外，其它的都要通过扫描和验证。每个 ROM 空间的开头两个字节一定是 0xaa55，称为“ROM 签名”，由宏操作 romsignature()加以检验。有的 ROM 空间只要存在就一定在某个特定的地址上，有的则可以在一个特定的区间中，但一定与 2KB 边界对齐。这种多样性是在长期的发展过程中形成的。

回到 setup_arch()的代码中，将母板上的 ROM 空间纳入 iomem_resource 树中以后，还要根据前面收集在 e820 数据结构中的信息，将由 BIOS 扫描探测到的各个内存区间也纳入 iomem_resource 树的统一管理（814～838 行）。对于 RAM 空间，这里还进一步分解出内核的代码段和数据段两片特殊用途的空间。此外，图形接口上也有一片 RAM 空间，其地址为 0xA0000～0xBFFFF（见 arch/i386/kernel/setup.c）。

==================== arch/i386/kernel/setup.c 321 323 ====================

```
static struct resource code_resource = { "Kernel code", 0x100000, 0 };
static struct resource data_resource = { "Kernel data", 0, 0 };
static struct resource vram_resource = { "Video RAM area", 0xa0000, 0xbffff, IORESOURCE_BUSY };
```

对于内核代码段和数据段中的地址信息，是由 setup_arch()在一开始时根据由 gcc 提供的_text、_etext、_edata 等信息设置的（见 634～637 行）。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

系统固有的 I/O 类资源则定义于数组 standard_io_resources[]中（arch/i386/kernel/setup.c），842～843 行的 for 循环将这些 resource 结构链入 ioport_resource 树中：

==================== arch/i386/kernel/setup.c 308 317 ====================
```
struct resource standard_io_resources[] = {
	{ "dma1", 0x00, 0x1f, IORESOURCE_BUSY },
	{ "pic1", 0x20, 0x3f, IORESOURCE_BUSY },
	{ "timer", 0x40, 0x5f, IORESOURCE_BUSY },
	{ "keyboard", 0x60, 0x6f, IORESOURCE_BUSY },
	{ "dma page reg", 0x80, 0x8f, IORESOURCE_BUSY },
	{ "pic2", 0xa0, 0xbf, IORESOURCE_BUSY },
	{ "dma2", 0xc0, 0xdf, IORESOURCE_BUSY },
	{ "fpu", 0xf0, 0xff, IORESOURCE_BUSY }
};
```

以结构中的第一项为例，I/O 地址 0x00～0x1f 共 32 个寄存器是用于第一个 DMA 控制器的。这里固定设置好的都是系统固有的资源，使用了 0x00～0xff 共 256 个寄存器。从 0x100 开始就是用于一般外部设备的 I/O 地址，需要在相应外部设备的初始化过程中向系统登记。

最后，一般 PC 机都配备 VGA 图形卡，所以使作为全局量的指针 conswitchp 指向数据结构 vga_con。这是 VGA 卡设备驱动层的函数跳转结构，提供了各种屏幕操作的函数指针。

在系统初始化的第二阶段，setup_arch()也许是最重要的函数，正如其函数所暗示的那样，它所设置的是系统的“ architecture”，即总体的结构。完成了 setup_arch()，内核就有了一个框架，内核中的各个部件或模块就有了运行的环境。

但是这个阶段的初始化远未完成，路还长得很，我们回到 start_kernel()中再往下看。

==================== init/main.c 532 545 ====================
[start_kernel()]
```
	printk("Kernel command line: %s\n", saved_command_line);
	parse_options(command_line);
	trap_init();
	init_IRQ();
	sched_init();
	time_init();
	softirq_init();

	/*
	 * HACK ALERT! This is early. We're enabling the console before
	 * we've done PCI setups etc, and console_init() must be aware of
	 * this. But we do want output early, in case something goes wrong.
	 */
	console_init();
```

这里调用的 init_IRQ()、softirq_init()以及 time_init()都已在第 3 章中看过了，读者不妨回过去温习一下。我们要看的是其余几个函数。

首先是对引导命令行中各个选择项的分析和处理。前面在 setup_arch()中（639 行）已经对命令行作过一些分析处理，但那只是专门针对与内存有关的选择项，现在则通过 parse_options()对命令行进行

PDF-XCHANGE Click to buy NOW! www.docu-track.com

全面的分析处理。函数 parse_options()所作的只是一些字符串的处理，目的在于从命令行中分解出各个选择项，然后对每个选择项调用 checksetup()加以处理。我们跳过 parse_options()而直接看 checksetup()，其代码在 init/main.c 中：

==================== init/main.c 318 332 ====================
[start_kernel()>parse_options()>checksetup()]

```
static int __init checksetup(char *line)
{
	struct kernel_param *p;

	p = &__setup_start;
	do {
		int n = strlen(p->str);
		if (!strncmp(line,p->str,n)) {
			if (p->setup_func(line+n))
				return 1;
		}
		p++;
	} while (p < &__setup_end);
	return 0;
}
```

内核中有个数组，数组中的每个元素都是一个 kernel_param 数据结构，定义于 include/linux/init.h：

==================== include/linux/init.h 56 62 ====================

```
/*
 * Used for kernel command line parameter setup
 */
struct kernel_param {
	const char *str;
	int (*setup_func)(char *);
};
```

根据由 parse_options()传下来的字符串，checksetup()在这个数组中逐个搜索比对，如果与某一选择项的字符串相符，就执行由相应 kernel_param 数据结构通过函数指针提供的操作。那么，数组中的数据结构是在哪儿定义的呢？首先，在 include/linux/init.h 中定义了一个宏操作__setup()：

==================== include/linux/init.h 66 70 ====================

```
#define __setup(str, fn)						\
	static char __setup_str_##fn[] __initdata = str;				\
	static struct kernel_param __setup_##fn __attribute__((unused)) __initsetup = { __setup_str_##fn, fn }

#endif /* __ASSEMBLY__ */
```

这里的__initdata 和__initsetup 分别定义为：

==================== include/linux/init.h 78 78 ====================

```
#define __initdata __attribute__ ((__section__ (".data.init")))
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== include/linux/init.h 80 80 ====================

```
#define __initsetup     __attribute__ ((unused,__section__ (".setup.init")))
```

表示在连接时要分别将有关的数据结构放在“.data.init”和“.setup.init”段中。

这样，以选择项“no387”为例，在 include/asm-i386/bugs.h 中有这么一行：

==================== include/asm-i386/bugs.h 51 51 ====================

```
__setup("no387", no_387);
```

经过 gcc 的预处理以后，这一行就变成了这样：

```
static char __setup_str_no_387[ ] __initdata = " no387";         \
static struct kernel_param __setup_no_387__attribute__((unused)) __initsetup =
    { __setup_str_no_387, no_387 }
```

于是，如果命令行中有“no387”这个选择项，checksetup()就会在字符串比对相符以后调用函数 no_387()，其代码也在 include/asm-i386/bugs.h 中：

==================== include/asm-i386/bugs.h 44 49 ====================

```
static int __init no_387(char *s)
{
     boot_cpu_data.hard_math = 0;
     write_cr0(0xE | read_cr0());
     return 1;
}
```

对这个选择项的处理是很简单的，只是把 boot_cpu_data.hard_math 设成 0，并把控制寄存器%cr0 中的几个标志位设成 1，表示系统中没有 80387 协处理器，或者即使有也不用。

再如选择项“root=”，表示要将文件系统的总根建立在指定的设备上，其定义见 init/main.c：

==================== init/main.c 316 316 ====================

```
__setup("root=", root_dev_setup);
```

相应的函数 root_dev_setup()则为（init/main.c）：

==================== init/main.c 299 314 ====================

```
static int __init root_dev_setup(char *line)
{
     int i;
     char ch;

     ROOT_DEV = name_to_kdev_t(line);
     memset (root_device_name, 0, sizeof root_device_name);
     if (strncmp (line, "/dev/", 5) == 0) line += 5;
     for (i = 0; i < sizeof root_device_name - 1; ++i)
     {
         ch = line[i];
         if ( isspace (ch) || (ch == ',') || (ch == '\0') ) break;
         root_device_name[i] = ch;
     }
```

```
        return 1;
}
```

显然，由 checksetup()传下来的字符串已经跳过了前面的“root=”，而剩下的部分正是 root_dev_setup()所关心的。代码本身很简单，我们就不加解释了。

这样的选择项有多少呢？有几十个之多。一般，如果内核中某个模块或设备驱动程序的设计人员觉得有需要提供一个引导选择项 xyz，就可以提供一个函数 do_xyz()，并在程序中加上一行“__setup("xyz", do_xyz);”，这就行了。

下一个重要的函数是 trap_init()，这个函数的作用是设置陷阱门和中断门，读者在第 3 章中已经看过它的代码。但是当时我们的注意力集中在中断机制，所以跳过了在 trap_init()中调用的一个函数 cpu_init()。

显然，cpu_init()与 CPU 的初始化有关。如前所述，CPU 本身的初始化主要是在 startup_32()中完成的，但是实际上有一部分层次比较高的初始化放在第二阶段才进行，这就是这里的 cpu_init()。与 startup_32()中的初始化相比，这里主要是为进程调度作准备，其代码在 arch/i386/kernel/setup.c 中：

==================== arch/i386/kernel/setup.c 2199 2266 ====================

[start_kernel()>trap_init()>cpu_init()]

```
/*
 * cpu_init() initializes state that is per-CPU. Some data is already
 * initialized (naturally) in the bootstrap process, such as the GDT
 * and IDT. We reload them nevertheless, this function acts as a
 * 'CPU state barrier', nothing should get across.
 */
void __init cpu_init (void)
{
        int nr = smp_processor_id();
        struct tss_struct * t = &init_tss[nr];

        if (test_and_set_bit(nr, &cpu_initialized)) {
                printk("CPU#%d already initialized!\n", nr);
                for (;;) __sti();
        }
        printk("Initializing CPU#%d\n", nr);

        if (cpu_has_vme || cpu_has_tsc || cpu_has_de)
                clear_in_cr4(X86_CR4_VME|X86_CR4_PVI|X86_CR4_TSD|X86_CR4_DE);
#ifndef CONFIG_X86_TSC
        if (tsc_disable && cpu_has_tsc) {
                printk("Disabling TSC...\n");
                /**** FIX-HPA: DOES THIS REALLY BELONG HERE? ****/
                clear_bit(X86_FEATURE_TSC, boot_cpu_data.x86_capability);
                set_in_cr4(X86_CR4_TSD);
        }
#endif

        __asm__ __volatile__("lgdt %0": "=m" (gdt_descr));
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
	__asm__ __volatile__("lidt %0": "=m" (idt_descr));

	/*
	 * Delete NT
	 */
	__asm__("pushfl ; andl $0xffffbfff,(%esp) ; popfl");

	/*
	 * set up and load the per-CPU TSS and LDT
	 */
	atomic_inc(&init_mm.mm_count);
	current->active_mm = &init_mm;
	if(current->mm)
		BUG();
	enter_lazy_tlb(&init_mm, current, nr);

	t->esp0 = current->thread.esp0;
	set_tss_desc(nr,t);
	gdt_table[__TSS(nr)].b &= 0xfffffdff;
	load_TR(nr);
	load_LDT(&init_mm);

	/*
	 * Clear all 6 debug registers:
	 */

#define CD(register) __asm__("movl %0,%%db" #register ::"r"(0) );

	CD(0); CD(1); CD(2); CD(3); /* no db4 and db5 */; CD(6); CD(7);

#undef CD

	/*
	 * Force FPU initialization:
	 */
	current->flags &= ~PF_USEDFPU;
	current->used_math = 0;
	stts();
}
```

这个函数不仅由主 CPU 在 trap_init()中调用，也由次 CPU 在进入 start_secondary()以后调用。代码中的 cpu_initialized 是个全局的位图，系统中的每个 CPU 在这个位图中都有个对应的标志位，一个 CPU 完成了本身的初始化就将相应的标志位设置成 1，让其它 CPU，其实主要是主 CPU，知道这个 CPU 的初始化已经完成了。对于每个具体的 CPU 而言，将 cpu_initialized 中的对应位设置成 1 只能发生一次。如果有哪一个 CPU 再次试图设置其对应标志位就肯定是出了问题，所以就让出了问题的 CPU 陷入一个无限循环。这实际上相当于让 CPU 跳过下面的所有操作进入空转，因为 CPU 在这个循环中仍能响应中断请求，并且在中断服务以后若发现需要调度就会如常进行调度。不过，CPU 在所谓“空转”中

会进入停机状态，实际上是不转，而在这个 for 循环中则真正在打转。

跳过对 TSC 等的设置，下面对段寄存器的设置、对（当前进程的）task_struct 结构中的指针 active_mm 的设置，以及为什么 task_struct 结构中的指针 mm 应该是 0，还有对“任务寄存器”的设置，结合第 3 章和第 4 章的内容读者应该能自己读懂。代码中的 2233 行将 CPU 中“标志位寄存器”的 NT 位（表示嵌套进入内核，“ Nested Task”）清成 0。由于没有直接对这个寄存器进行位操作的指令，这里先把它压入堆栈，在堆栈中操作完了再送回寄存器。2248 行通过 load_LDT()装入局部段描述表指针 LDTR，不过 Linux 内核只在 VM86 模式下才使用局部段，所以我们在这里并不关心。还有，2256 行是对 CPU 中一些程序调试寄存器的初始化，2263～2265 行是对浮点处理器的初始化，这里就都从略了。

再看对进程调度机制的初始化。

==================== kernel/sched.c 1244 1269 ====================

[start_kernel()>sched_init()]

```
void __init sched_init(void)
{
	/*
	 * We have to do a little magic to get the first
	 * process right in SMP mode.
	 */
	int cpu = smp_processor_id();
	int nr;

	init_task.processor = cpu;

	for(nr = 0; nr < PIDHASH_SZ; nr++)
		pidhash[nr] = NULL;

	init_timervecs();

	init_bh(TIMER_BH, timer_bh);
	init_bh(TQUEUE_BH, tqueue_bh);
	init_bh(IMMEDIATE_BH, immediate_bh);

	/*
	 * The boot idle thread does lazy MMU switching as well:
	 */
	atomic_inc(&init_mm.mm_count);
	enter_lazy_tlb(&init_mm, current, cpu);
}
```

这里所调用的一些函数对读者已经不陌生了。虽然还没有看过 init_timervecs()的代码，但是顾名思义这是对定时器中断向量的初始化。所以，我们把这个函数留给读者。

然后是对系统主控终端的初始化，读者已经在“设备驱动”一章中看到过了。

我们还得往下看。

==================== init/main.c 546 562 ====================

[start_kernel()]

```
#ifdef CONFIG_MODULES
    init_modules();
#endif
    if (prof_shift) {
        unsigned int size;
        /* only text is profiled */
        prof_len = (unsigned long) &_etext - (unsigned long) &_stext;
        prof_len >>= prof_shift;

        size = prof_len * sizeof(unsigned int) + PAGE_SIZE-1;
        prof_buffer = (unsigned int *) alloc_bootmem(size);
    }

    kmem_cache_init();
    sti();
    calibrate_delay();
#ifdef CONFIG_BLK_DEV_INITRD
. . . . . .
```

==================== init/main.c 569 596 ====================

```
#endif
    mem_init();
    kmem_cache_sizes_init();
#ifdef CONFIG_3215_CONSOLE
      con3215_activate();
#endif
#ifdef CONFIG_PROC_FS
    proc_root_init();
#endif
    mempages = num_physpages;

    fork_init(mempages);
    proc_caches_init();
    vfs_caches_init(mempages);
    buffer_init(mempages);
    page_cache_init(mempages);
    kiobuf_setup();
    signals_init();
    bdev_init();
    inode_init(mempages);
#if defined(CONFIG_SYSVIPC)
    ipc_init();
#endif
#if defined(CONFIG_QUOTA)
    dquot_init_hash();
#endif
    check_bugs();
    printk("POSIX conformance testing by UNIFIX\n");
```

首先是对可安装模块这个机制的初始化，其实只是计算出内核符号表的大小，而且计算也很简单。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

函数 init_modules()的代码在 kernel/module.c 中：

==================== kernel/module.c 229 237 ====================
[start_kernel()>init_modules()]

```
/*
 * Called at boot time
 */

void __init init_modules(void)
{
	kernel_module.nsyms = __stop___ksymtab - __start___ksymtab;

#ifdef __alpha__
. . . . . .
```

==================== kernel/module.c 239 240 ====================

```
#endif
}
```

物理页面 empty_zero_page 在前面被借用来传递引导命令行和参数块，现在应该物归原主了。顾名思义，这个页面的内容应该是全 0。此外，基本内存中的页面，即 1MB 以下的页面还没有释放以供动态分配，前面使用的页面位图所在的页面现在也已经完成了使命，也应该释放以供动态分配。还有，前面讲过，在页面映射目录中的低区，即对应于用户空间的区间有几个临时性的目录项，最后应该予以清除。同时，还需要收集或计算出一些统计信息。这些操作都是由 mem_init()完成的，其代码在 arch/i386/mm/init.c 中：

==================== arch/i386/mm/init.c 554 630 ====================
[start_kernel()>mem_init()]

```
void __init mem_init(void)
{
	int codesize, reservedpages, datasize, initsize;
	int tmp;

	if (!mem_map)
		BUG();

#ifdef CONFIG_HIGHMEM
	highmem_start_page = mem_map + highstart_pfn;
	max_mapnr = num_physpages = highend_pfn;
#else
	max_mapnr = num_physpages = max_low_pfn;
#endif
	high_memory = (void *) __va(max_low_pfn * PAGE_SIZE);

	/* clear the zero-page */
	memset(empty_zero_page, 0, PAGE_SIZE);

	/* this will put all low memory onto the freelists */
	totalram_pages += free_all_bootmem();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```

	reservedpages = 0;
	for (tmp = 0; tmp < max_low_pfn; tmp++)
		/*
		 * Only count reserved RAM pages
		 */
		if (page_is_ram(tmp) && PageReserved(mem_map+tmp))
			reservedpages++;
#ifdef CONFIG_HIGHMEM
	for (tmp = highstart_pfn; tmp < highend_pfn; tmp++) {
		struct page *page = mem_map + tmp;

		if (!page_is_ram(tmp)) {
			SetPageReserved(page);
			continue;
		}
		ClearPageReserved(page);
		set_bit(PG_highmem, &page->flags);
		atomic_set(&page->count, 1);
		__free_page(page);
		totalhigh_pages++;
	}
	totalram_pages += totalhigh_pages;
#endif
	codesize =  (unsigned long) &_etext - (unsigned long) &_text;
	datasize =  (unsigned long) &_edata - (unsigned long) &_etext;
	initsize =  (unsigned long) &__init_end - (unsigned long) &__init_begin;

	printk("Memory: %luk/%luk available (%dk kernel code, %dk reserved, %dk data, %dk init, %ldk highmem)\n",
		(unsigned long) nr_free_pages() << (PAGE_SHIFT-10),
		max_mapnr << (PAGE_SHIFT-10),
		codesize >> 10,
		reservedpages << (PAGE_SHIFT-10),
		datasize >> 10,
		initsize >> 10,
		(unsigned long) (totalhigh_pages << (PAGE_SHIFT-10))
		 );

#if CONFIG_X86_PAE
	if (!cpu_has_pae)
		panic("cannot execute a PAE-enabled kernel on a PAE-less CPU!");
#endif
	if (boot_cpu_data.wp_works_ok < 0)
		test_wp_bit();

	/*
	 * Subtle. SMP is doing it's boot stuff late (because it has to
	 * fork idle threads) - but it also needs low mappings for the
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
623         * protected-mode entry to work. We zap these entries only after
624         * the WP-bit has been tested.
625         */
626  #ifndef CONFIG_SMP
627         zap_low_mappings();
628  #endif
629
630  }
```

这里的 571 行将 empty_zero_page 的内容清成全 0。接着调用的 free_all_bootmem()是个重要的操作，它根据页面位图的内容释放物理内存中所有可供动态分配的页面，将这些页面投入动态分配。

==================== mm/bootmem.c 300 303 ====================

[start_kernel()>mem_init()>free_all_bootmem()]

```
300  unsigned long __init free_all_bootmem (void)
301  {
302         return(free_all_bootmem_core(&contig_page_data));
303  }
```

==================== mm/bootmem.c 224 261 ====================

[start_kernel()>mem_init()>free_all_bootmem()>free_all_bootmem_core()]

```
224  static unsigned long __init free_all_bootmem_core(pg_data_t *pgdat)
225  {
226         struct page *page = pgdat->node_mem_map;
227         bootmem_data_t *bdata = pgdat->bdata;
228         unsigned long i, count, total = 0;
229         unsigned long idx;
230
231         if (!bdata->node_bootmem_map) BUG();
232
233         count = 0;
234         idx = bdata->node_low_pfn - (bdata->node_boot_start >> PAGE_SHIFT);
235         for (i = 0; i < idx; i++, page++) {
236                if (!test_bit(i, bdata->node_bootmem_map)) {
237                       count++;
238                       ClearPageReserved(page);
239                       set_page_count(page, 1);
240                       __free_page(page);
241                }
242         }
243         total += count;
244
245         /*
246          * Now free the allocator bitmap itself, it's not
247          * needed anymore:
248          */
249         page = virt_to_page(bdata->node_bootmem_map);
250         count = 0;
251         for (i = 0; i < ((bdata->node_low_pfn-(bdata->node_boot_start >> PAGE_SHIFT))/8 +
PAGE_SIZE-1)/PAGE_SIZE; i++,page++) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
252             count++;
253             ClearPageReserved(page);
254             set_page_count(page, 1);
255             __free_page(page);
256         }
257         total += count;
258         bdata->node_bootmem_map = NULL;
259
260         return total;
261     }
```

这里的 bdata->node_boot_start 是在前面的 init_bootmem_core()中设置的，表示整个物理内存的起点（页面号），其数值为 0，但是在特殊的情况下也可以不是 0。而 bdata->node_low_pfn 则为物理内存中最高的页面号（不包括 HIGHMEM）。235～242 行的 for 循环扫描整个物理内存的所有页面，对于每个页面都测试页面位图中的对应位，如果对应位为 0 就表示这个页面可以投入分配。把相应 page 结构中的使用计数设置成 1，假装这个页面已经分配，再对其调用__free_page()，就把这个 page 结构挂入了所属的空闲页面队列。可以说，前面为物理页面管理机制和空闲页面队列的建立作了很多的准备，特别是页面位图的设立，最终为的就是这一步。正因为这样，一旦完成了这一步，页面位图就不再需要，因而它所占据的物理页面也可以释放了，251～256 行的 for 循环就是用来释放这些页面。

在 mem_init()中还有条件地调用了一个函数 test_wp_bit()，这是针对 80386 CPU 而设的，现在已经没有多大意义了。注意 627 行对 zap_low_mappings()的调用是条件编译的语句，仅对单处理器系统有效。在初始化的第一阶段建立起了页面映射目录，为了向页式映射过渡，当时的目录中同时在系统空间和用户空间映射了物理内存的开头 8MB 空间。后来把系统空间的映射扩展到了整个物理内存，但是其用户空间的映射尚未拆除。而内核线程在正常运行中是不应该使用 0xC0000000 以下的地址的，所以应该把用户空间的映射拆除。然而，在 SMP 结构的系统中，现在次 CPU 尚未开始运行，当次 CPU 开始运行时也要经历初始化的第一阶段，也要向页式映射过渡，所以现在还不能拆除这些映射。

回到 start_kernel()的代码中，kmem_cache_sizes_init()是对内核中通用 slab 缓冲区管理机制的初始化，通过 kmem_cache_create()建立起大小分别为 32 字节、64 字节、…、128KB 的缓冲区队列，这些 slab 缓冲区是供内核通过 kmalloc()动态分配的，专用的 slab 缓冲区则不在其中。接着，proc_root_init()是对特殊文件系统/proc 的初始化，读者已经在第 5 章看过这个函数的代码。再下面是 fork_init()，根据物理内存的大小计算出允许创建线程（包括进程）的数量，其代码在 kernel/fork.c 中：

==================== kernel/fork.c 66 77 ====================

[start_kernel()>fork_init()]

```
void __init fork_init(unsigned long mempages)
{
	/*
	 * The default maximum number of threads is set to a safe
	 * value: the thread structures can take up at most half
	 * of memory.
	 */
	max_threads = mempages / (THREAD_SIZE/PAGE_SIZE) / 2;

	init_task.rlim[RLIMIT_NPROC].rlim_cur = max_threads/2;
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        init_task.rlim[RLIMIT_NPROC].rlim_max = max_threads/2;
}
```

接下来的 proc_caches_init()、vfs_caches_init()、buffer_init()以及 kiobuf_setup()和 signals_init()，还有 bdev_init()和 inode_init()，基本上都是为有关的管理机制建立起专用 slab 缓冲区队列。而 page_cache_init()则分配空间建立起缓冲页面杂凑表 page_hash_table，这个函数的代码在 mm/filemap.c 中：

==================== mm/filemap.c 2583 2608 ====================

[start_kernel()>page_cache_ink()]

```
void __init page_cache_init(unsigned long mempages)
{
    unsigned long htable_size, order;

    htable_size = mempages;
    htable_size *= sizeof(struct page *);
    for(order = 0; (PAGE_SIZE << order) < htable_size; order++)
        ;

    do {
        unsigned long tmp = (PAGE_SIZE << order) / sizeof(struct page *);

        page_hash_bits = 0;
        while((tmp >>= 1UL) != 0UL)
            page_hash_bits++;

        page_hash_table = (struct page **)
            __get_free_pages(GFP_ATOMIC, order);
    } while(page_hash_table == NULL && --order > 0);

    printk("Page-cache hash table entries: %d (order: %ld, %ld bytes)\n",
          (1 << page_hash_bits), order, (PAGE_SIZE << order));
    if (!page_hash_table)
        panic("Failed to allocate page hash table\n");
    memset((void *)page_hash_table, 0, PAGE_HASH_SIZE * sizeof(struct page *));
}
```

这里引用的宏操作定义于 include/linux/pagemap.h：

==================== include/linux/pagemap.h 42 43 ====================

```
#define PAGE_HASH_BITS (page_hash_bits)
#define PAGE_HASH_SIZE (1 << PAGE_HASH_BITS)
```

缓冲页面杂凑表是个 page 结构指针数组，数组中的每一项都用来维持一个 page 结构的单链队列。这个杂凑表的使用对于提高文件操作和页面换入／换出的效率起着重要的作用。杂凑表的大小是 2 的整数次幂，大的杂凑表有利于使页面在队列中更趋分散，从而有利于提高查找效率，但是太大，也没有必要。所以，这里根据物理内存的大小先估算出一个期望值，然后试着按这个大小分配空间，如果分配不到就降格以求，减小对大小的要求。最后，成功分配到所需的空间以后，就把它初始化成全 0。

PDF-XCHANGE Click to buy NOW! www.docu-track.com

至于 ipc_init()，那当然是对 Sys V 进程间通信机制的初始化。

==================== ipc/util.c 34 40 ====================

[start_kernel()>ipc_init()]

```
void __init ipc_init (void)
{
	sem_init();
	msg_init();
	shm_init();
	return;
}
```

这些函数大体上都一样，都是对有关数据结构如 msg_ids、shm_ids 的初始化，同时为通过/proc 特殊文件系统获取有关信息创造条件，在/proc 目录下建立起“ sysvipc/shm”等节点。

初始化的第二阶段终于接近尾声了，下面是最后的冲刺。

==================== init/main.c 597 608 ====================

[start_kernel()]

```

	/*
	 *	We count on the initial thread going ok
	 *	Like idlers init is an unlocked kernel thread, which will
	 *	make syscalls (and thus be locked).
	 */
	smp_init();
	kernel_thread(init, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGNAL);
	unlock_kernel();
	current->need_resched = 1;
	cpu_idle();
}
```

迄今为止的初始化一直都只有一个 CPU 在执行（尽管有些代码是公用的）。如果是 SMP 结构的系统，就一直只有主 CPU 在运行，其余的 CPU 都是“次 CPU”，都要受到主 CPU 的启动才能运行。现在，系统的“经济基础”已经建立，可以启动这些次 CPU 的运行了。这里的 smp_init()就是对 SMP 结构的初始化，具体包括启动系统中的各个次 CPU，让它们完成自身的初始化并进入各自的空转进程。读者已经在前一章中看到这个函数的代码以及对有关过程的介绍。

从 smp_init()返回到 start_kernel()时，SMP 结构的初始化已经完成，所有的次 CPU 都已经或者即将经由 start_secondary()进入 cpu_idle()。主 CPU 在初始化的第二阶段还有一件事要做，那就是创建内核线程 init()。创建了这个内核线程以后，主 CPU 也会转入 cpu_idle()。可见，不管是主 CPU 还是次 CPU，最终都会进入 cpu_idle()。当系统中所有的 CPU 都进入 cpu_idle()中时，它们就站在了同一个起点上，开始公平竞争了。这个函数的代码在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 117 139 ====================

[start_kernel()>cpu_idle()]

```
/*
 * The idle thread. There's no useful work to be
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
 * done, so just try to conserve power and have a
 * low exit latency (ie sit in a loop waiting for
 * somebody to say that they'd like to reschedule)
 */
void cpu_idle (void)
{
    /* endless idle loop with no priority at all */
    init_idle();
    current->nice = 20;
    current->counter = -100;

    while (1) {
        void (*idle)(void) = pm_idle;
        if (!idle)
            idle = default_idle;
        while (!current->need_resched)
            idle();
        schedule();
        check_pgt_cache();
    }
}
```

这个函数的主体是个无穷循环，一进入这个循环就再也出不来了。既然所有的 CPU 都在执行这个循环，而且在循环中也并不引用与此前的历史有关的变量，也就分不出什么主次，所以从此以后系统中所有的 CPU 就都互相平等了。当一个 CPU 在 cpu_idle()中执行时，就是处于该 CPU 的“空转”进程中。读者在前面已经看到，空转进程并不挂入系统中的就绪队列，其 task_struct 结构指针保持在以 CPU 编号为下标的数组 init_tasks[]中。由于空转进程不挂入就绪队列，在进程调度时就永远不会以空转进程为目标。相反，只有在没有进程可以运行时才会回到空转进程中来。

那么，CPU 在空转进程中干些什么呢？简言之就是循环地执行一个空转函数，直到进程的 task_struct 结构中的 need_resched 变成 1，此时就通过 schedule()进行一次进程调度。CPU 一进入 schedule()，就一直要到系统中再无就绪进程可以运行时才会返回到 cpu_idle()中，然后就再等待 current->need_resched 变成 1。一般情况下的空转函数为 default_idle()，其代码见 arch/i386/kernel/process.c（在引导时可以在命令行中通过选择项“idle=poll”将 pm_idle 设置成指向 poll_idle()，但是两个函数并无本质的不同）。

==================== arch/i386/kernel/process.c 80 89 ====================

[cpu_idle()>default_idle()]

```
static void default_idle(void)
{
    if (current_cpu_data.hlt_works_ok && !hlt_counter) {
        __cli();
        if (!current->need_resched)
            safe_halt();
        else
            __sti();
    }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
}
```

==================== include/asm-i386/system.h 300 301 ====================

```
/* used in the idle loop; sti takes one instruction cycle to complete */
#define safe_halt()         __asm__ __volatile__("sti; hlt": : :"memory")
```

当 CPU 在 cpu_idle()中，而 task_struct 结构中的 need_resched 又为 0 时，就是实在无事可干了，所以把中断打开，并通过“hlt”指令进入“停机”状态，或者说硬件睡眠状态。处于这个状态的 CPU 不执行任何指令，直到有中断请求到来时才会恢复运行，那时就好像刚结束 hlt 指令的执行一样。然后，当 CPU 结束对中断的处理时，就会回到 hlt 指令的后继指令，那就是从 default_idle()返回了。取决于具体的中断，在中断服务程序中有可能会使当前进程的 need_resched 标志变成 1，那样就会在 cpu_idle()中结束 134 行的 while 循环而调用 schedule()，否则就又进入 default_idle()。

当系统中所有的 CPU 都进入 cpu_idle()中时，就绪队列中只有一个进程，那就是前面 604 行创建的 init()。所以，实际上只有一个 CPU 能在 schedule()中调度这个惟一的进程运行，因此只让一个 CPU 执行 schedule()就行了。让谁执行 schedule()呢？“近水楼台先得月”，原先的主 CPU 在放下身份与次 CPU“打成一片”之前留了一手，在 606 行把它自己的 need_resched 标志设成了 1，从而成了最先进行调度的 CPU，当仁不让地取得了执行 init()的权利。而其它的 CPU，则自会在 cpu_idle()中的 134 行受到阻挡而暂时不能前进。

10.4 系统初始化（第三阶段）

内核线程 init()的任务仍然还是初始化，当然是进一步的、更高层次上的初始化。事实上，从引导结束，CPU 转入内核映象开始，一共有三个阶段的初始化。第一个阶段是从进入 startup_32()开始到进入 start_kernel()或者 start_secondary()。这个阶段主要是对 CPU 自身的初始化，主 CPU 和次 CPU 都要经历这种初始化，但是主 CPU 要多作一些贡献。第二个阶段是从进入 start_kernel()开始到进入 cpu_idle()。这个阶段主要是对系统的“经济基础”，即各种资源的初始化，仅由主 CPU 进行。第三个阶段则是 init()的执行，这是对系统的“上层建筑”的初始化。表面上此时已无主 CPU 和次 CPU 之分，由谁执行 init()取决于竞争调度的结果，但是由于主 CPU 预先留了一手，实际上总是由它执行的。函数 init()的代码在 init/main.c 中，这个函数本身并不长，但是实际进行的操作却不简单，所以我们还是要分段阅读。

==================== init/main.c 761 773 ====================

```
static int init(void * unused)
{
	lock_kernel();
	do_basic_setup();

	/*
	 * Ok, we have completed the initial bootup, and
	 * we're essentially up and running. Get rid of the
	 * initmem segments and start the user-mode stuff..
	 */
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        free_initmem();
        unlock_kernel();

```

这里在对内核加锁的条件下调用了 do_basic_setup()和 free_initmem()两个函数。表面上此时系统中只有一个进程可以被调度运行，因而只有一个 CPU 能够运行，似乎不必加锁。但是，下面就会看到，在执行 init()的过程中还会创建新的进程。另一方面，如果发生中断，则这些 CPU 还是有可能在中断服务程序中造成干扰。

先看 do_basic_setup()，这是主要的，其代码在 init/main.c 中。我们删去了代码中一些条件编译的片段，另一些条件编译的片段因为本来就只有一行语句而没有删去，但是我们在这里基本上不关心这些选择项，有兴趣或需要的读者可自行研读。这些（我们不关心的）条件编译选择项有：

- CONFIG_BLK_DEV_INITRD。用于 RAMDISK，即以一部分内存来模拟硬盘。
- CONFIG_MTRR。如果 CPU 中有 MTRR 寄存器，就可以通过这个寄存器按区间来管理内存的高速缓冲。不过，也可以不用 MTRR，而按页面来管理高速缓冲。
- CONFIG_SYSCTL。允许在运行中动态地改变一些内核中的参数。
- CONFIG_SBUS。SBUS 是 Sparc 工作站中采用的总线。
- CONFIG_PPC。适用于 Power PC 处理器。
- CONFIG_MCA。用于 PS/2 系统结构中的 Micro Channel 外设总线。
- CONFIG_ARCH_ACORN。适用于 ARM 处理器。
- CONFIG_ZORRO。一种总线。
- CONFIG_DIO。一种总线。
- CONFIG_NUBUS。Macintosh 计算机中的总线。
- CONFIG_ISAPNP。用于 ISA 总线上支持 PnP（Plug and Play）的接口卡。
- CONFIG_TC。用于一种称为 Turbo Channel 的总线。
- CONFIG_IRDA。用于红外线通信接口。
- CONFIG_PCMCIA。用于笔记本电脑的外插接口卡。

至于 CONFIG_PCI，则虽然也是选择项（笔记本电脑中没有 PCI 总线），但是实际上不但用于桌上型 PC，也广泛用于嵌入式系统中，读者已经在“设备驱动”一章中看过其初始化函数 pci_init()的代码。

==================== init/main.c 641 650 ====================

[init()>do_basic_setup()]

```
/*
 * Ok, the machine is now initialized. None of the devices
 * have been touched yet, but the CPU subsystem is up and
 * running, and memory and process management works.
 *
 * Now we can finally start doing some real work..
 */
static void __init do_basic_setup(void)
{
#ifdef CONFIG_BLK_DEV_INITRD
. . . . . .
```

==================== init/main.c 652 715 ====================

```
#endif

        /*
         * Tell the world that we're going to be the grim
         * reaper of innocent orphaned children.
         *
         * We don't want people to have to make incorrect
         * assumptions about where in the task array this
         * can be found.
         */
        child_reaper = current;

#if defined(CONFIG_MTRR)    /* Do this after SMP initialization */
/*
 * We should probably create some architecture-dependent "fixup after
 * everything is up" style function where this would belong better
 * than in init/main.c..
 */
        mtrr_init();
#endif

#ifdef CONFIG_SYSCTL
        sysctl_init();
#endif

        /*
         * Ok, at this point all CPU's should be initialized, so
         * we can start looking into devices..
         */
#ifdef CONFIG_PCI
        pci_init();
#endif
#ifdef CONFIG_SBUS
        sbus_init();
#endif
#if defined(CONFIG_PPC)
        ppc_init();
#endif
#ifdef CONFIG_MCA
        mca_init();
#endif
#ifdef CONFIG_ARCH_ACORN
        ecard_init();
#endif
#ifdef CONFIG_ZORRO
        zorro_init();
#endif
#ifdef CONFIG_DIO
```

```
        dio_init();
#endif
#ifdef CONFIG_NUBUS
        nubus_init();
#endif
#ifdef CONFIG_ISAPNP
        isapnp_init();
#endif
#ifdef CONFIG_TC
        tc_init();
#endif

        /* Networking initialization needs a process context */
        sock_init();

#ifdef CONFIG_BLK_DEV_INITRD
 . . . . . .
```

==================== init/main.c 720 740 ====================

```
#endif

        start_context_thread();
        do_initcalls();

        /* .. filesystems .. */
        filesystem_setup();

#ifdef CONFIG_IRDA
        irda_device_init(); /* Must be done after protocol initialization */
#endif
#ifdef CONFIG_PCMCIA
        init_pcmcia_ds();       /* Do this last */
#endif

        /* Mount the root filesystem.. */
        mount_root();

        mount_devfs_fs ();

#ifdef CONFIG_BLK_DEV_INITRD
 . . . . . .
```

==================== init/main.c 758 759 ====================

```
#endif
}
```

跳过条件编译部分，剩下也就不多了。我们把 sock_init()的代码留给读者结合第 7 章自己阅读，这里先看 start_context_thread()的代码，它在 kernel/context.c 中：

==================== kernel/context.c 149 153 ====================

[init()>do_basic_setup()>start_context_thread()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
int start_context_thread(void)
{
	kernel_thread(context_thread, NULL, CLONE_FS | CLONE_FILES | CLONE_SIGHAND);
	return 0;
}
```

这是系统中创建的第二个内核线程，是一个所谓“守护神”（daemon），或者说“代理人”，名叫“ keventd”。其代码 context_thread()也在 kernel/context.c 中：

==================== kernel/context.c 66 108 ====================

```
static int context_thread(void *dummy)
{
	struct task_struct *curtask = current;
	DECLARE_WAITQUEUE(wait, curtask);
	struct k_sigaction sa;

	daemonize();
	strcpy(curtask->comm, "keventd");
	keventd_running = 1;
	keventd_task = curtask;

	spin_lock_irq(&curtask->sigmask_lock);
	siginitsetinv(&curtask->blocked, sigmask(SIGCHLD));
	recalc_sigpending(curtask);
	spin_unlock_irq(&curtask->sigmask_lock);

	/* Install a handler so SIGCLD is delivered */
	sa.sa.sa_handler = SIG_IGN;
	sa.sa.sa_flags = 0;
	siginitset(&sa.sa.sa_mask, sigmask(SIGCHLD));
	do_sigaction(SIGCHLD, &sa, (struct k_sigaction *)0);

	/*
	 * If one of the functions on a task queue re-adds itself
	 * to the task queue we call schedule() in state TASK_RUNNING
	 */
	for (;;) {
		set_task_state(curtask, TASK_INTERRUPTIBLE);
		add_wait_queue(&context_task_wq, &wait);
		if (TQ_ACTIVE(tq_context))
			set_task_state(curtask, TASK_RUNNING);
		schedule();
		remove_wait_queue(&context_task_wq, &wait);
		run_task_queue(&tq_context);
		wake_up(&context_task_done);
		if (signal_pending(curtask)) {
			while (waitpid(-1, (unsigned int *)0, __WALL|WNOHANG) > 0)
				;
			flush_signals(curtask);
```

```
                recalc_sigpending(curtask);
            }
        }
}
```

这个进程的主体是一个无穷 for 循环，平时在一个队列 context_task_wq 中睡眠等待。在一些设备驱动程序中，有一些函数可能需要在一个进程的上下文中执行（而不是在中断处理的上下文中执行），keventd 就为这种函数充当着“代理人”的角色。需要把一个函数提交给 keventd，使这个函数得以在 keventd 的上下文中执行时，就为这个函数准备下一个 tq_struct 数据结构，然后通过 schedule_task()把这个数据结构挂入队列 tq_context，并唤醒在队列 context_task_wq 中睡眠的 keventd。当 keventd 被调度运行时，就会以它的名义，在它的上下文中通过 run_task_queue()执行挂在 tq_context 中的函数。函数 schedule_task()的代码在 kernel/context.c 中，读者可与 context_thread()的代码对照着阅读。

==================== kernel/context.c 44 64 ====================

```
/**
 * schedule_task - schedule a function for subsequent execution in process context.
 * @task: pointer to a &tq_struct which defines the function to be scheduled.
 *
 * May be called from interrupt context.  The scheduled function is run at some
 * time in the near future by the keventd kernel thread.  If it can sleep, it
 * should be designed to do so for the minimum possible time, as it will be
 * stalling all other scheduled tasks.
 *
 * schedule_task() returns non-zero if the task was successfully scheduled.
 * If @task is already residing on a task queue then schedule_task() fails
 * to schedule your task and returns zero.
 */
int schedule_task(struct tq_struct *task)
{
    int ret;
    need_keventd(__FUNCTION__);
    ret = queue_task(task, &tq_context);
    wake_up(&context_task_wq);
    return ret;
}
```

那么，这种函数与第 3 章中的 bh 函数有什么区别呢？我们讲过，bh 函数实质上是中断服务程序的一部分，是在中断服务的上下文中执行的，因而受到一些限制，而在进程上下文中执行的函数就不受这些限制。例如，在 bh 函数中就不应该有可能受阻而需要睡眠等待的操作，而在进程上下文中执行的函数则可以使当前进程，即 keventd 线程进入睡眠。

从 context_thread()的代码中还可以看出，一般的内核线程要调用一个函数 daemonize()才能变成“守护神”，这个函数的作用其实只是释放从父进程继承下来的一些资源（斩断尘缘才能成神），而改换门庭投到 init_task 的门下，共享它的资源。

==================== kernel/sched.c 1197 1226 ====================

[context_thread()>daemonize()]

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
/*
 *   Put all the gunge required to become a kernel thread without
 *   attached user resources in one place where it belongs.
 */

void daemonize(void)
{
        struct fs_struct *fs;


        /*
         * If we were started as result of loading a module, close all of the
         * user space pages.  We don't need them, and if we didn't close them
         * they would be locked into memory.
         */
        exit_mm(current);

        current->session = 1;
        current->pgrp = 1;

        /* Become as one with the init task */

        exit_fs(current);  /* current->fs->count--; */
        fs = init_task.fs;
        current->fs = fs;
        atomic_inc(&fs->count);
        exit_files(current);
        current->files = init_task.files;
        atomic_inc(&current->files->count);
}
```

同到 do_basic_setup()，调用的下一个函数是 do_initcalls()，其代码在 init/main.c 中：

==================== init/main.c 627 639 ====================

[init()>do_basic_setup()>do_initcalls()]

```
627  static void __init do_initcalls(void)
628  {
629         initcall_t *call;
630
631         call = &__initcall_start;
632         do {
633              (*call)();
634              call++;
635         } while (call < &__initcall_end);
636
637         /* Make sure there is no pending stuff from the initcall sequence */
638         flush_scheduled_tasks();
639  }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这个函数貌不惊人，可是干的事情却非同小可！

大家都知道，内核映象也跟一般可执行程序的映象一样有代码段和数据段。但是这只是一般而言，事实上内核映象中还有其它一些比较小的段和子段。内核源代码中有个文件 arch/i386/vmlinux.lds 是对连接工具 GNU ld 的连接说明，文件中描述了内核映象中所有的段和子段，读者不妨看一下。我们在这里关心的是其中的两个，一个是“ .text.init”、我们称之为“初始化代码段”，另一个则是“ .initcall.init”、我们称之为“初始化调用段”。有关的定义如下（include/linux/init.h）：

==================== include/linux/init.h 43 49 ====================

```
/*
 * Used for initialization calls..
 */
typedef int (*initcall_t)(void);
typedef void (*exitcall_t)(void);

extern initcall_t __initcall_start, __initcall_end;
. . . . . .
```

==================== include/linux/init.h 51 52 ====================

```
#define __initcall(fn)                                  \
        static initcall_t __initcall_##fn __init_call = fn
. . . . . .
```

==================== include/linux/init.h 76 76 ====================

```
#define __init          __attribute__ ((__section__ (".text.init")))
. . . . . .
```

==================== include/linux/init.h 81 81 ====================

```
#define __init_call     __attribute__ ((unused,__section__ (".initcall.init")))
```

内核的代码中，凡是只在系统初始化时调用一次，以后就不会再受到调用的函数就在前面加上一个限定词__init，例如上面的 do_initcalls()本身就是这样的一个函数。这些函数在编译、连接以后就全都集中在一起，都在“初始化代码段”中。之所以要这样，是因为这些函数所占的内存空间在初始化完成以后就可以另作它用，而集中在一起就可以成片地回收。

“初始化代码段”中的函数不是一律平等的，其中有些函数是某一方面初始化的“入口函数”，所以需要在系统初始化时从 do_initcalls()中加以调用，而同在初始化代码段中的另一些函数则自会因此而辗转地受到调用。对“入口函数”的调用通常都是独立的，并不互相依赖，所以孰先孰后没有什么关系。可是怎样调用呢？当然可以按常规把对这些函数的调用全都列出在 do_initcalls()中。但是这样多少会给内核代码的编写和维护带来一些不便，例如每要增加或减少一种什么功能时就得相应地修改 do_initcalls()的代码。为了减少这种不便，内核代码的作者们采用了一个巧妙的办法，那就是为这些需要在初始化时加以调用的函数都准备了一个函数指针，并把这些函数指针都集中在一起形成一个“初始化调用段”，实际上就是一个函数指针数组。这样，就可以通过一个循环来调用所有这些函数了，这就是 do_initcalls()中的 do-while 循环的来历，代码中的__initcall_start 和__initcall_end 分别是这个段的起点和终点。这样的安排为编程带来了方便。例如，假定你要在内核中增加一个功能模块 xyz，并且需要静态地连入内核映象中（有些基本的功能不一定适合采用动态安装模块），而所有的程序都在一个文件 xyz.c 中，初始化入口为 xyz_init()，则只要在同一文件中加上一行“ __initcall(xyz_init);”就可以了，并不需要到 do_initcalls()的代码中去作什么修改。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

那么，这样的函数指针都有哪一些呢？我们在这里只能略举数例。一个是 mm/slab.c 中定义的 mm_cpucache_init()，这是 SMP 结构中对 stab 缓冲块队列的一些优化，需要作为初始化程序来执行（mm/slab.c）。

==================== mm/slab.c 471 471 ====================

```
__initcall(kmem_cpucache_init);
```

经过 gcc 的预处理，就会在初始化调用段中生成出一个指向 kmem_cpucache_init()的函数指针 __initcall_kmem_cpucache_init：

```
static initcall_t __initcall_kmem_cpucache_init
   __attribute__ ((unused, __section__ (".initcall.init "))) = kmem_cpucache_init;
```

再如 drivers/pci/proc.c 中定义的 pci_proc_init()，这是对特殊文件系统/proc 中子目录/proc/pci 的初始化，用于 PCI 总线的管理。

==================== drivers/pci/proc.c 437 437 ====================

```
__initcall(pci_proc_init);
```

经过 gcc 的预处理就会生成一个指向 pci_proc_init()的函数指针__initcall_pci_proc_init。不过，这些都不是我们所关心的。我们在这里关心的是在 fs/partitions/check.c 中的一个函数 partition_setup()：

==================== fs/partitions/check.c 456 456 ====================

```
__initcall(partition_setup);
```

不知道代码的作者为什么给它取了这么个名字，其实这个函数与磁盘的分区毫无关系，实际上处理的是外部设备的初始化，这是系统初始化的一个重要组成部分。

==================== fs/partitions/check.c 442 454 ====================

[init()>do_basic_setup()>do_initcalls()>partition_setup()]

```
int __init partition_setup(void)
{
      device_init();

#ifdef CONFIG_BLK_DEV_RAM
#ifdef CONFIG_BLK_DEV_INITRD
      if (initrd_start && mount_initrd) initrd_load();
      else
#endif
      rd_load();
#endif
      return 0;
}
```

函数的主体（我们忽略条件编译部分）是 device_init()，其代码在 drivers/block/genhd.c 中：

==================== drivers/block/genhd.c 34 67 ====================

[init()>do_basic_setup()>do_initcalls()>partition_setup()>device_init()]

```
void __init device_init(void)
```

```
{
#ifdef CONFIG_PARPORT
     parport_init();
#endif
     chr_dev_init();
     blk_dev_init();
     sti();
#ifdef CONFIG_I2O
     i2o_init();
#endif
#ifdef CONFIG_BLK_DEV_DAC960
     DAC960_Initialize();
#endif
#ifdef CONFIG_FC4_SOC
     /* This has to be done before scsi_dev_init */
     soc_probe();
#endif
#ifdef CONFIG_IEEE1394
       ieee1394_init();
#endif
#ifdef CONFIG_BLK_CPQ_DA
     cpqarray_init();
#endif
#ifdef CONFIG_NET
     net_dev_init();
#endif
#ifdef CONFIG_ATM
     (void) atmdev_init();
#endif
#ifdef CONFIG_VT
     console_map_init();
#endif
}
```

可见，这是所有外部设备初始化的总入口，里面包括了块设备的初始化 blk_dev_init()、一般字符设备的初始化 chr_dev_init()，还有一些特殊设备的初始化，如并行口的 parport_init()、网络设备的 net_dev_init()和 atmdev-init()等等。其中的有些初始化程序已经在“设备驱动”一章中读过，有些（网络设备）不在本书内容的范围之内，有些则虽在本书内容的范围之内但尚未读过。但是，限于本节篇幅，我们别无选择，只好把这些函数留给读者了。特别地，我们建议读者结合“设备驱动”一章读一下 blk_dev_init()中调用的 ide_init()，那是对 IDE 硬盘的初始化。

事实上，所有静态模块的初始化都是通过__initcall()说明的，只不过是间接的，通过另一个宏定义 module_init()转了个弯而已（见 include/linux/init.h）：

==================== include/linux/init.h 89 89 ====================

```
#define module_init(x) __initcall(x);
```

例如，对 Ext2 文件系统的支持就是以模块的形式实现的，所以在 fs/ext2/super.c 中就有这么一行：

PDF-XCHANGE Click to buy NOW! www.docu-track.com

==================== fs/ext2/super.c 800 800 ====================

```
module_init(init_ext2_fs)
```

所以函数 init_ext2_fs()也是由 do_initcalls()调用的，实际上其代码也在初始化代码段中：

==================== fs/ext2/super.c 788 791 ====================
[init()>do_basic_setup()>do_initcalls()>init_ext2_fs()]

```
static int __init init_ext2_fs(void)
{
        return register_filesystem(&ext2_fs_type);
}
```

显然，到 do_initcalls()执行完毕的时候，所有此类初始化函数都已经执行过了，内核中由静态模块支持的所有文件系统也都已向系统登记。

回到 do_basic_setup()的代码中，下面（726 行）是对 filesystem_setup()的调用。不过，这里所谓“文件系统”只是指几个特殊文件系统，主要是 devfs。其它还有作为条件编译选择项的“网络文件系统”nfs 以及对 Unix96 PTY“伪终端”的支持，这些特殊文件系统的安装并不依赖于文件系统的根设备。其中 devfs 的安装是在安装根设备之前的准备工作，如果系统支持 devfs，则 devfs 的初始安装对于在引导命令行中采用“ root=”选择项有帮助。不过，这里对 devfs 的安装只是“预安装”，就是通过 kern_mount()进行的安装，以后还要通过 do_mount()再“重安装”一次。

==================== fs/filesystems.c 28 39 ====================
[init()>do_basic_setup()>filesystem_setup()]

```
void __init filesystem_setup(void)
{
	init_devfs_fs();  /*  Header file may make this empty  */

#ifdef CONFIG_NFS_FS
	init_nfs_fs();
#endif

#ifdef CONFIG_DEVPTS_FS
	init_devpts_fs();
#endif
}
```

现在到了安装文件系统根设备的时候了，根设备的安装不同于此后其它设备的安装。一般设备（文件系统）的安装都要先通过/dev 目录找到代表着待安装设备的节点，从而取得待安装设备的设备号。而安装点一般（除一些特殊文件系统外）也是在已安装文件系统中的某个节点。安装根设备时/dev 目录尚不存在，但是一般根设备就是引导设备，其设备号已经在参数块中传递了过来，而根设备的安装点“/”本来就不在已经安装的文件系统中，不需要寻找。不过，如果在引导命令行中指定了以另一个设备作为根设备，那就需要一些起码的根据设备名找到设备号的能力，这就是在此之前先预安装 devfs 的目的。此外还有一点不同之处，常规的安装都事先知道设备上是哪一种文件系统（见 do_mount()的参数），而根设备的安装却并不事先知道文件系统的类型，而需要在安装时试凑。所以，内核中有个函数 mount_root()专用于根设备的安装，其代码在 fs/super.c 中。这个函数比较长，所以我们分段阅读，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

同时我们也从代码中删去了一些条件编译的片段。

==================== fs/super.c 1462 1474 ====================

[init()>do_basic_setup()>mount_root()]

```
void __init mount_root(void)
{
    struct file_system_type * fs_type;
    struct super_block * sb;
    struct vfsmount *vfsmnt;
    struct block_device *bdev = NULL;
    mode_t mode;
    int retval;
    void *handle;
    char path[64];
    int path_start = -1;

#ifdef CONFIG_ROOT_NFS
    . . . . . .
```

==================== fs/super.c 1507 1509 ====================

```
#endif

#ifdef CONFIG_BLK_DEV_FD
    . . . . . .
```

==================== fs/super.c 1529 1541 ====================

```
#endif

    devfs_make_root (root_device_name);
    handle = devfs_find_handle (NULL, ROOT_DEVICE_NAME,
                                MAJOR (ROOT_DEV), MINOR (ROOT_DEV),
                          DEVFS_SPECIAL_BLK, 1);
    if (handle)  /*  Sigh: bd*() functions only paper over the cracks  */
    {
        unsigned major, minor;

        devfs_get_maj_min (handle, &major, &minor);
        ROOT_DEV = MKDEV (major, minor);
    }
```

代码中的 root_device_name[]是个全局量，如果在引导命令行中使用了“root=”选择项，就会在一个初始化函数 root_dev_setup()中从这个选择项中将设备名的前缀“/dev/”去掉，然后复制到这个字符串中。函数 devfs_make_root()把旧式的设备名如“ide/hd/0”等等转换成新式的设备名，其一般形式为“/host0/bus1/target2/lun3/part4”。然后，便通过 devfs_find_handle()在 devfs 文件系统中寻找，这里用到了一些宏操作定义或常数定义（见 fs/super.c 及 include/linux/devfs_fs_kernel.h）：

==================== include/linux/devfs_fs_kernel.h 48 52 ====================

```
#ifdef CONFIG_BLK_DEV_INITRD
#  define ROOT_DEVICE_NAME ((real_root_dev ==ROOT_DEV) ? root_device_name:NULL)
#else
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
51   #  define ROOT_DEVICE_NAME root_device_name
52   #endif
```

==================== fs/super.c 59 59 ====================

```
59   kdev_t ROOT_DEV;
```

ROOT_DEV 看起来像是个常数，实际上却是个全局变量。这个变量的初值是在系统初始化的第一阶段在 setup_arch()中设置的，实际上来自引导辅助程序和 BIOS。一般而言，从什么设备上引导，那个设备就是根设备，除非通过“root=”选择项另加指定。读者也许感到困惑：系统是由 BIOS 引导的，而“设备号”是 Unix/Linux 所特有的，就算 BIOS 从 IDE 接口上的某个磁盘或盘区引导，它又怎么能知道这个设备在 Linux 中的设备号是什么呢？其实，BIOS（或者 LILO）只是从引导设备上读入了引导扇区，并且把引导扇区在内存中的映象当作可执行代码而跳转到它的开头处，以后的事就取决于引导扇区的内容和辅助程序 setup 了。而这二者是与具体的系统密切相关的，实际上就是系统的一部分，而且就存放在具体的设备上，当然就能知道这设备的设备号是什么了。

所以，代码中的 1531～1541 行是为通过“root=”选择项改变根设备而设的。如果在引导命令行中没有使用这个选择项，或者内核不支持 devfs，那就保持 ROOT_DEV 的初值不变。

==================== fs/super.c 1542 1581 ====================

[init()>do_basic_setup()>mount_root()]

```
1542
1543      /*
1544       * Probably pure paranoia, but I'm less than happy about delving into
1545       * devfs crap and checking it right now. Later.
1546       */
1547      if (!ROOT_DEV)
1548           panic("I have no root and I want to scream");
1549
1550      bdev = bdget(kdev_t_to_nr(ROOT_DEV));
1551      if (!bdev)
1552           panic(__FUNCTION__ ": unable to allocate root device");
1553      bdev->bd_op = devfs_get_ops (handle);
1554      path_start = devfs_generate_path (handle, path + 5, sizeof (path) - 5);
1555      mode = FMODE_READ;
1556      if (!(root_mountflags & MS_RDONLY))
1557           mode |= FMODE_WRITE;
1558      retval = blkdev_get(bdev, mode, 0, BDEV_FS);
1559      if (retval == -EROFS) {
1560           root_mountflags |= MS_RDONLY;
1561           retval = blkdev_get(bdev, FMODE_READ, 0, BDEV_FS);
1562      }
1563      if (retval) {
1564              /*
1565            * Allow the user to distinguish between failed open
1566            * and bad superblock on root device.
1567            */
1568           printk ("VFS: Cannot open root device \"%s\" or %s\n",
1569                root_device_name, kdevname (ROOT_DEV));
1570           printk ("Please append a correct \"root=\" boot option\n");
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1571            panic("VFS: Unable to mount root fs on %s",
1572                kdevname(ROOT_DEV));
1573        }
1574
1575        check_disk_change(ROOT_DEV);
1576        sb = get_super(ROOT_DEV);
1577        if (sb) {
1578            fs_type = sb->s_type;
1579            goto mount_it;
1580        }
1581
```

知道了根设备的设备号，下面的操作就与普通的安装没有多大不同了，引为设备驱动层已经在此之前完成了初始化，根据设备号就可以找到有关的数据结构。例如，1550 行的 bdget()根据设备号找到或者创建给定设备的 block_device 数据结构，读者已经在“设备驱动”一章中读过它的代码。1553 行的 devfs_get_ops()找到设备的 block_device_operations 数据结构。然后，blkdev_get()通过这个数据结构中提供的 open 操作“打开”目标设备。这里先在 1558 行试着按可写模式打开，如果失败就在 1561 行再按只读模式打开。至于 1576 行的 get_super()，则扫描已经读入系统中的超级块队列。如果在这个队列中找到了目标设备上的超级块，就可以知道该设备上的文件系统是什么类型的（1578 行），从而可以进入真正的“安装”操作（1579 行）。当然，对于根设备的安装，其超级块还不在超级队列中，因而需要从设备上读入。

我们接着往下看。

==================== fs/super.c 1582 1597 ====================

[init()>do_basic_setup()>mount_root()]

```
1542
1543        /*
1544         * Probably pure paranoia, but I'm less than happy about delving into
1545         * devfs crap and checking it right now. Later.
1546         */
1547        if (!ROOT_DEV)
1548            panic("I have no root and I want to scream");
1549
1550        bdev = bdget(kdev_t_to_nr(ROOT_DEV));
1551        if (!bdev)
1552            panic(__FUNCTION__ ": unable to allocate root device");
1553        bdev->bd_op = devfs_get_ops (handle);
1554        path_start = devfs_generate_path (handle, path + 5, sizeof (path) - 5);
1555        mode = FMODE_READ;
1556        if (!(root_mountflags & MS_RDONLY))
1557            mode |= FMODE_WRITE;
1558        retval = blkdev_get(bdev, mode, 0, BDEV_FS);
1559        if (retval == -EROFS) {
1560            root_mountflags |= MS_RDONLY;
1561            retval = blkdev_get(bdev, FMODE_READ, 0, BDEV_FS);
1562        }
1563        if (retval) {
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
1564            /*
1565             * Allow the user to distinguish between failed open
1566             * and bad superblock on root device.
1567             */
1568            printk ("VFS: Cannot open root device \"%s\" or %s\n",
1569                root_device_name, kdevname (ROOT_DEV));
1570            printk ("Please append a correct \"root=\" boot option\n");
1571            panic("VFS: Unable to mount root fs on %s",
1572                kdevname(ROOT_DEV));
1573        }
1574
1575        check_disk_change(ROOT_DEV);
1576        sb = get_super(ROOT_DEV);
1577        if (sb) {
1578            fs_type = sb->s_type;
1579            goto mount_it;
1580        }
1581
```

内核中的 file_systems 是一个 file_system_type 数据结构队列，队列中的每个数据结构都代表着一种受到内核支持的文件系统。队列中的各个数据结构都通过函数指针提供其自己的 read_super 操作。这里依次以各种文件系统的 file_system_type 数据结构为参数调用 read_super()，实际上就是让各种文件系统都来试试，看谁能从设备上读入一个格式相符的超级块。读者已经在“文件系统”一章中读过 read_super()的代码。

读入了超级块，就可以进入安装阶段了。

==================== fs/super.c 1598 1619 ====================

[init()>do_basic_setup()>mount_root()]

```
1598 mount_it:
1599        printk ("VFS: Mounted root (%s filesystem)%s.\n",
1600            fs_type->name,
1601            (sb->s_flags & MS_RDONLY) ? " readonly" : "");
1602        if (path_start >= 0) {
1603            devfs_mk_symlink (NULL, "root", DEVFS_FL_DEFAULT,
1604                        path + 5 + path_start, NULL, NULL);
1605            memcpy (path + path_start, "/dev/", 5);
1606            vfsmnt = add_vfsmnt(NULL, sb->s_root, path + path_start);
1607        }
1608        else
1609            vfsmnt = add_vfsmnt(NULL, sb->s_root, "/dev/root");
1610        /* FIXME: if something will try to umount us right now... */
1611        if (vfsmnt) {
1612            set_fs_root(current->fs, vfsmnt, sb->s_root);
1613            set_fs_pwd(current->fs, vfsmnt, sb->s_root);
1614            if (bdev)
1615                bdput(bdev); /* sb holds a reference */
1616            return;
1617        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
        panic("VFS: add_vfsmnt failed for root fs");
}
```

函数 add_vfsmnt()的代码已经在“文件系统”一章中读过，这里就不多说了。函数 set_fs_root()设置当前进程的 fs_struct 数据结构中的 root 和 rootmnt 两个指针，使它们分别指向根节点的 dentry 数据结构，就是这里的 sb->s_root，以及安装根设备时的“连接件 ” vfsmount 数据结构，就是这里的 vfsmnt。同时，set_fs_pwd()将当前进程的“当前工作目录”，即 pwd 和 pwdmnt 两个指针也设置成指向这两个数据结构。读者将会看到，现在的这个当前进程就是系统中所有进程的始祖，把这个进程的这些指针设置好了，以后由它 fork()出来的进程就都会继承这些指针，以整个文件系统的根为本进程逻辑意义上的“根”，并且以此为本进程的当前工作目录。

完成了根设备的安装以后，如果需要的话，就可以进一步完成特殊文件系统 devfs 的安装。在安装根设备之前，已经通过 kern_mount()将 devfs 预安装了一次。但是，我们以前提到过，有些文件系统需要在此后再“重安装”一次，这样才能与具体文件系统中的某个具体的节点挂上钩。所以 do_basic_setup()接着就调用 mount_devfs_fs()来做这件事。

==================== fs/devfs/base.c 3363 3371 ====================

[init()>do_basic_setup()>mount_devfs_fs()]

```
void __init mount_devfs_fs (void)
{
    int err;

    if ( (boot_options & OPTION_NOMOUNT) ) return;
    err = do_mount ("none", "/dev", "devfs", 0, "");
    if (err == 0) printk ("Mounted devfs on /dev\n");
    else printk ("Warning: unable to mount devfs, err: %d\n", err);
}   /*  End Function mount_devfs_fs  */
```

可见，如果内核支持 devfs，并且需要安装，那就把它安装在/dev 节点上。

至此，do_basic_setup()的执行已经完成。回到 init()的代码中，初始化代码段中的函数都已经执行过了。如前所述，这些函数都只需要在系统初始化时执行一次，现在既然已经完成，就可以“过河拆桥”，回收它们所占的空间另作它用了。函数 free_initmem()的作用就是逐个页面地回收整个初始化代码段的内存空间，其代码在 arch/i386/mm/init.c 中：

==================== arch/i386/mm/init.c 656 668 ====================

[init()>free_initmem()]

```
void free_initmem(void)
{
        unsigned long addr;

        addr = (unsigned long)(&__init_begin);
        for (; addr < (unsigned long)(&__init_end); addr += PAGE_SIZE) {
            ClearPageReserved(virt_to_page(addr));
            set_page_count(virt_to_page(addr), 1);
            free_page(addr);
            totalram_pages++;
        }
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
    printk ("Freeing unused kernel memory: %dk freed\n", (&__init_end - &__init_begin) >> 10);
}
```

至此，内核的初始化已经完成，下面要再往上跑一层，处理应用层的初始化，为应用程序的运行，首先是系统管理作好准备了。我们接着往下看 init()的代码。

==================== init/main.c 774 794 ====================

[init()]

```
    if (open("/dev/console", O_RDWR, 0) < 0)
        printk("Warning: unable to open an initial console.\n");

    (void) dup(0);
    (void) dup(0);

    /*
     * We try each of these until one succeeds.
     *
     * The Bourne shell can be used instead of init if we are
     * trying to recover a really broken machine.
     */

    if (execute_command)
        execve(execute_command,argv_init,envp_init);
    execve("/sbin/init",argv_init,envp_init);
    execve("/etc/init",argv_init,envp_init);
    execve("/bin/init",argv_init,envp_init);
    execve("/bin/sh",argv_init,envp_init);
    panic("No init found.  Try passing init= option to kernel.");
}
```

这里的 dup()和 execve()都是系统调用。当前进程 init()是个运行于系统空间的内核线程，虽然也能作系统调用，却不能像在用户空间那样通过普通的 C 语言库函数进行，而要由内核中的系统调用入口函数，我们以 execve()为例来看这些函数的定义（include/asm-i386/unistd.h）：

==================== include/asm-i386/unistd.h 273 282 ====================

```
#define _syscall3(type,name,type1,arg1,type2,arg2,type3,arg3) \
type name(type1 arg1,type2 arg2,type3 arg3) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
    : "=a" (__res) \
    : "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
          "d" ((long)(arg3))); \
__syscall_return(type,__res); \
}
```

==================== include/asm-i386/unistd.h 233 240 ====================

```
#define __syscall_return(type, res) \
do { \
    if ((unsigned long)(res) >= (unsigned long)(-125)) { \
```

```
        errno = -(res); \
        res = -1; \
    } \
    return (type) (res); \
} while (0)
```

宏操作_syscall3()用于所有带三个参数的系统调用，而 execve()就是这样的系统调用：

==================== include/asm-i386/unistd.h 342 342 ====================

```
static inline _syscall3(int,execve,const char *,file,char **,argv,char **,envp)
```

经过 gcc 的预处理以后，就生成了函数 execve()的定义：

```
int execve(const char *file, char **argv, char **envp)
{
  long  res;
  __asm__ volatile ("int $0x80"
     : "=a" (__res)
     : "0" (__NR_execve), "b" ((long)(file)), "c" ((long)(argv)),
         "d" ((long)(envp)));

  do {
      if ((unsigned long)(res) >= (unsigned long)(-125)) {
         errno=  -(res);
         res = -1;
      }
      return (int) (res);
  } while (0)
}
```

对这段代码可能需要一些解释。这里的输入部说明把参数 file 放在寄存器%ebx 中，把参数 argv 放在寄存器%ecx 中，把参数 envp 放在寄存器%edx 中。系统调用号__NR_execve 则放在操作数 0 中，那就是局部变量 res。由于 res 是局部量，所以是在堆栈中，而且在这里是在堆栈顶部。所有这些正好构成了执行指令“ int 0x80”进入系统的条件。同时，输入部又说明 res 既是局部变量（因而在堆栈中），又跟寄存器%eax 结合，因为系统调用通过%eax 返回执行的结果。

需要在内核中调用的其它系统调用，如 dup()、open()等等，都是按同样的方式生成的。内核中相应地还有用于不带参数、带 1 个参数、带 2 个参数等等的宏操作_syscall0()、_syscall1()、syscall2()等等。

前面的代码中先打开/dev/console，由于这是当前进程第一次打开文件，其打开文件号必定为 0。接着调用 dup()把由打开文件号 0 所代表的连接复制两次，就使打开文件号 0、1 和 2 都成为同一个连接的代表，这就是通常所说的“标准输入”、“标准输出”以及“标准出错信息”三个通道 stdin、stdout 以及 stderr。打开了这三个标准 I/O 通道以后，就可以像普通在 shell 下启动的进程那样，通过系统调用 execve()执行各种可执行程序了。读者在第 4 章中看到，execve()是条不归之路，一旦执行目标程序成功，最后就从目标程序直接 exit()了。但是，如果执行目标程序失败，例如在文件系统中找不到目标程序，那就会从 execve()失败返回。所以，这里实际上是依次尝试执行/sbin/init、/etc/init、/bin/init、/bin/sh。一般来说，至少/bin/sh 是一定可以执行成功的，否则系统就根本无法运行了。

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

那么，/sbin/init 又干些什么呢？主要是根据文件/etc/inittab 的规定分叉（创建）出一些进一步初始化的进程，包括对/etc/rc.d/rc 的执行，不过那已经不属于内核的范围了，读者可以通过命令“ man 8 init ”阅读对这个程序的说明，或阅读有关 Linux 系统管理的资料。分叉出来的进程中还包括若干执行/sbin/getty 的进程，这些进程后来就变成了执行/bin/login 或/sbin/sulogin 的进程。就这样，这个以系统初始化为开始的进程就成为整个系统中所有进程共同的“祖宗”，这个进程以后一直留在系统中，就是系统的 1 号进程。

总之，系统中有几个处理器就有几个 0 号进程，那就是各个处理器的空转进程，这些进程都不在进程的杂凑队列中，也不在就绪进程队列中，不参与进程调度。但是，1 号进程却只有一个，这是系统中所有进程的祖宗，从这个进程开始的所有进程都参与调度。

10.5　系统的关闭和重引导

最后我们再看看关闭系统的过程。Linux 为此提供了一个系统调用 reboot()，内核中的实现为 sys_reboot()。顾名思义，这个系统调用的功能应该是“重引导”，但是实际上这是个多功能的系统调用。根据参数 cmd 的值，这个系统调用可以用于以下一些目的：

- “重引导”（LINUX_REBOOT_CMD_RESTART 或 LINUX_REBOOT_CMD_RESTART2）。
- “停机”（LINUX_REBOOT_CMD_HALT）。
- “关机”（LINUX_REBOOT_CMD_POWER_OFF）。

此外，还可以用来设置键组 C_A_D，即 Ctrl-Alt-Delete 三键同按时的作用。在 DOS 和 Windows 操作系统上同时按这三个键表示重引导（在 Windows NT 上则又不是），所以 Linux 允许用户选择用或不用这个特殊的组合。

在为重引导或停机而调用 reboot()之前，应用进程应该先调用另一个系统调用 sync()，以保证把文件系统中所有已经改变了的缓冲区和数据结构的内容先写入磁盘。

函数 sys_reboot()的代码在 kernel/sys.c 中：

==================== kernel/sys.c 261 331 ====================

```
/*
 * Reboot system call: for obvious reasons only root may call it,
 * and even root needs to set up some magic numbers in the registers
 * so that some mistake won't make this reboot the whole machine.
 * You can also set the meaning of the ctrl-alt-del-key here.
 *
 * reboot doesn't sync: do that yourself before calling this.
 */
asmlinkage long sys_reboot(int magic1, int magic2, unsigned int cmd, void * arg)
{
     char buffer[256];

     /* We only trust the superuser with rebooting the system. */
     if (!capable(CAP_SYS_BOOT))
          return -EPERM;
```

PDF-XCHANGE
Click to buy NOW!
www.docu-track.com
PDF-XCHANGE
Click to buy NOW!
www.docu-track.com

```

	/* For safety, we require "magic" arguments. */
	if (magic1 != LINUX_REBOOT_MAGIC1 ||
	   (magic2 != LINUX_REBOOT_MAGIC2 && magic2 != LINUX_REBOOT_MAGIC2A &&
			magic2 != LINUX_REBOOT_MAGIC2B))
		return -EINVAL;

	lock_kernel();
	switch (cmd) {
	case LINUX_REBOOT_CMD_RESTART:
		notifier_call_chain(&reboot_notifier_list, SYS_RESTART, NULL);
		printk(KERN_EMERG "Restarting system.\n");
		machine_restart(NULL);
		break;

	case LINUX_REBOOT_CMD_CAD_ON:
		C_A_D = 1;
		break;

	case LINUX_REBOOT_CMD_CAD_OFF:
		C_A_D = 0;
		break;

	case LINUX_REBOOT_CMD_HALT:
		notifier_call_chain(&reboot_notifier_list, SYS_HALT, NULL);
		printk(KERN_EMERG "System halted.\n");
		machine_halt();
		do_exit(0);
		break;

	case LINUX_REBOOT_CMD_POWER_OFF:
		notifier_call_chain(&reboot_notifier_list, SYS_POWER_OFF, NULL);
		printk(KERN_EMERG "Power down.\n");
		machine_power_off();
		do_exit(0);
		break;

	case LINUX_REBOOT_CMD_RESTART2:
		if (strncpy_from_user(&buffer[0], (char *)arg, sizeof(buffer) - 1) < 0) {
			unlock_kernel();
			return -EFAULT;
		}
		buffer[sizeof(buffer) - 1] = '\0';

		notifier_call_chain(&reboot_notifier_list, SYS_RESTART, buffer);
		printk(KERN_EMERG "Restarting system with command '%s'.\n", buffer);
		machine_restart(buffer);
		break;

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
325     default:
326         unlock_kernel();
327         return -EINVAL;
328     }
329     unlock_kernel();
330     return 0;
331 }
```

在 sys_reboot()所实现的这些功能中，以重引导最为复杂，所以我们在这里读一下与此有关的代码，其余的就留给读者了。前面讲过，系统引导时可以使用一个带选择项的命令行，也可以不使用命令行，所以 sys_reboot()也为重引导提供了两个不同的命令码。我们只看不用命令行的情景，就是代码中的286～289 行。

内核中的有些模块（通常是外部设备）可能要求在关闭系统或重引导之前执行一些特殊的操作，例如有些老的硬盘就要求在断电之前先把磁头移到一个特定的“起降”磁道上。所以，要提供一种手段，使这样的模块在关闭系统或重引导之前能得到通知，执行一个预定的函数。这就是代码中调用 notifier_call_chain()的目的。如果一个模块要求在关闭系统或重引导之前执行一个函数，就可以准备好一个 notifier_block 数据结构，并向系统登记。这种数据结构的定义在 include/linux/notifier.h 中：

==================== include/linux/notifier.h 14 19 ====================

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

结构中的函数指针就用于需要在关闭系统或重引导之前执行的函数。准备好 notifier_block 数据结构以后，就可以通过 register_reboot_notifier()向系统登记（kernel/sys.c）：

==================== kernel/sys.c 149 152 ====================

```
int register_reboot_notifier(struct notifier_block * nb)
{
    return notifier_chain_register(&reboot_notifier_list, nb);
}
```

在 sys_reboot()中，当要关闭系统或重引导系统的时候，就通过 notifier_call_chain()执行已经登记的函数，调用的参数之一是一个表示调用原因的代码。

==================== kernel/sys.c 105 136 ====================

[sys_reboot()>notifier_call_chain()]

```
/**
 *  notifier_call_chain - Call functions in a notifier chain
 *  @n: Pointer to root pointer of notifier chain
 *  @val: Value passed unmodified to notifier function
 *  @v: Pointer passed unmodified to notifier function
 *
 *  Calls each function in a notifier chain in turn.
```

```
 *
 *	If the return value of the notifier can be and'd
 *	with %NOTIFY_STOP_MASK, then notifier_call_chain
 *	will return immediately, with the return value of
 *	the notifier function which halted execution.
 *	Otherwise, the return value is the return value
 *	of the last notifier function called.
 */

int notifier_call_chain(struct notifier_block **n, unsigned long val, void *v)
{
	int ret=NOTIFY_DONE;
	struct notifier_block *nb = *n;

	while(nb)
	{
		ret=nb->notifier_call(nb,val,v);
		if(ret&NOTIFY_STOP_MASK)
		{
			return ret;
		}
		nb=nb->next;
	}
	return ret;
}
```

调用了已经登记的函数以后，就可以通过 machine_restart()重引导了，这个函数的代码在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 346 375 ====================
[sys_reboot()>machine_restart()]

```
void machine_restart(char * __unused)
{
#if CONFIG_SMP
	/*
	 * Stop all CPUs and turn off local APICs and the IO-APIC, so
	 * other OSs see a clean IRQ state.
	 */
	smp_send_stop();
	disable_IO_APIC();
#endif

	if(!reboot_thru_bios) {
		/* rebooting needs to touch the page at absolute addr 0 */
		*((unsigned short *)__va(0x472)) = reboot_mode;
		for (;;) {
			int i;
			for (i=0; i<100; i++) {
				kb_wait();
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
364                     udelay(50);
365                     outb(0xfe,0x64);        /* pulse reset low */
366                     udelay(50);
367                 }
368                 /* That didn't work - force a triple fault.. */
369                 __asm__ __volatile__("lidt %0": :"m" (no_idt));
370                 __asm__ __volatile__("int3");
371             }
372         }
373
374         machine_real_restart(jump_to_bios, sizeof(jump_to_bios));
375     }
```

对于 SMP 结构的系统，先要通过 smp_send_stop()向系统中的其它 CPU 发送一个请求停机的处理器间中断请求，并关闭本 CPU 的“高级中断控制器 ” APIC。此外，执行 sys_reboot()的 CPU 还负有关闭全局的外部 APIC 的责任。然后，就要进行实际的“重启动”，即重引导了。

对系统的重启动可以通过两种不同的方法进行，一种是通过 BIOS 进行，另一种就是直接在系统中制造一次“总清”（reset）。具体采用哪一种方式取决于系统引导时是否使用了“ root=”选择项。如果没有使用这个选择项，那么全局量 reboot_thru_bios 为 0，所以会采用所谓“硬启动”制造一次总清。具体的过程见 360～371 行，我们就不深入到这些细节中去了，有兴趣或需要的读者可以结合 PC 和 i386 的有关技术资料自行阅读。

如果在引导命令行中使用了“ root=”选择项，则系统会在初始化的过程中执行一个函数 reboot_setup()，根据选择项的内容设置 reboot_thru_bios 和 reboot_mode 两个全局量的值，作为执行 reboot()系统调用时的依据。

==================== arch/i386/kernel/process.c 157 182 ====================

```
static int __init reboot_setup(char *str)
{
    while(1) {
        switch (*str) {
        case 'w': /* "warm" reboot (no memory testing etc) */
            reboot_mode = 0x1234;
            break;
        case 'c': /* "cold" reboot (with memory testing etc) */
            reboot_mode = 0x0;
            break;
        case 'b': /* "bios" reboot by jumping through the BIOS */
            reboot_thru_bios = 1;
            break;
        case 'h': /* "hard" reboot by toggling RESET and/or crashing the CPU */
            reboot_thru_bios = 0;
            break;
        }
        if((str = strchr(str,',')) != NULL)
            str++;
        else
```

```
                break;
        }
        return 1;
}

__setup("reboot=", reboot_setup);
```

如果在命令行中选择了通过 BIOS 重引导，则又有“热引导”和“冷引导”之分，二者都通过 machine_real_restart()进行。然而，不管是“热引导”还是“冷引导”，总之都要进入 BIOS。这里所谓进入 BIOS 并不是指像“ int 0x13”那样的 BIOS 调用，而是要进入整个 BIOS 的初始入口，就好像机器刚加电一样。以前我们提到过，BIOS 的入口是 0xffff0，读者可能觉得这很容易，只要执行一条 jmp 指令跳转到这个地址就行了。可是，事情并不是这么简单，BIOS 是为 16 位实地址模式设计的，而 CPU 此刻运行于 32 位保护模式，而且采用页式地址映射，这中间有着不小的差距。当初，次 CPU 在受到启动后是经过了一段“跳板”程序才从实地址模式进入保护模式；而主 CPU 则先执行了一段引导辅助程序，实际上也是“跳板”，才进入保护模式的。而且，二者在进入 startup_32()以后又经历了向页式映射的过渡。现在要回到 BIOS 就得走过相反的过程，上台靠跳板，下台要有台阶。而 machine_real_restart()的作用正是让当前 CPU 通过台阶下台。这个函数的代码在 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 254 344 ====================

[sys_reboot()>machine_restart()>machine_real_restart()]

```
/*
 * Switch to real mode and then execute the code
 * specified by the code and length parameters.
 * We assume that length will aways be less that 100!
 */
void machine_real_restart(unsigned char *code, int length)
{
        unsigned long flags;

        cli();

        /* Write zero to CMOS register number 0x0f, which the BIOS POST
           routine will recognize as telling it to do a proper reboot.  (Well
           that's what this book in front of me says -- it may only apply to
           the Phoenix BIOS though, it's not clear).  At the same time,
           disable NMIs by setting the top bit in the CMOS address register,
           as we're about to do peculiar things to the CPU.  I'm not sure if
           `outb_p' is needed instead of just `outb'.  Use it to be on the
           safe side.  (Yes, CMOS_WRITE does outb_p's. -  Paul G.)
         */

        spin_lock_irqsave(&rtc_lock, flags);
        CMOS_WRITE(0x00, 0x8f);
        spin_unlock_irqrestore(&rtc_lock, flags);

        /* Remap the kernel at virtual address zero, as well as offset zero
           from the kernel segment.  This assumes the kernel segment starts at
```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
   virtual address PAGE_OFFSET. */

memcpy (swapper_pg_dir, swapper_pg_dir + USER_PGD_PTRS,
     sizeof (swapper_pg_dir [0]) * KERNEL_PGD_PTRS);

/* Make sure the first page is mapped to the start of physical memory.
   It is normally not mapped, to trap kernel NULL pointer dereferences. */

pg0[0] = _PAGE_RW | _PAGE_PRESENT;

/*
 * Use `swapper_pg_dir' as our page directory.
 */
asm volatile("movl %0,%%cr3": :"r" (__pa(swapper_pg_dir)));

/* Write 0x1234 to absolute memory location 0x472.  The BIOS reads
   this on booting to tell it to "Bypass memory test (also warm
   boot)".  This seems like a fairly standard thing that gets set by
   REBOOT.COM programs, and the previous reset routine did this
   too. */

*((unsigned short *)0x472) = reboot_mode;

/* For the switch to real mode, copy some code to low memory.  It has
   to be in the first 64k because it is running in 16-bit mode, and it
   has to have the same physical and virtual address, because it turns
   off paging.  Copy it near the end of the first page, out of the way
   of BIOS variables. */

memcpy ((void *) (0x1000 - sizeof (real_mode_switch) - 100),
     real_mode_switch, sizeof (real_mode_switch));
memcpy ((void *) (0x1000 - 100), code, length);

/* Set up the IDT for real mode. */

__asm__ __volatile__ ("lidt %0" : : "m" (real_mode_idt));

/* Set up a GDT from which we can load segment descriptors for real
   mode.  The GDT is not used in real mode; it is just needed here to
   prepare the descriptors. */

__asm__ __volatile__ ("lgdt %0" : : "m" (real_mode_gdt));

/* Load the data segment registers, and thus the descriptors ready for
   real mode.  The base address of each segment is 0x100, 16 times the
   selector value being loaded here.  This is so that the segment
   registers don't have to be reloaded after switching to real mode:
   the values are consistent for real mode operation already. */

```

PDF-XCHANGE Click to buy NOW! www.docu-track.com

PDF-XCHANGE Click to buy NOW! www.docu-track.com

```
330      __asm__ __volatile__ ("movl $0x0010,%%eax\n"
331                    "\tmovl %%eax,%%ds\n"
332                    "\tmovl %%eax,%%es\n"
333                    "\tmovl %%eax,%%fs\n"
334                    "\tmovl %%eax,%%gs\n"
335                    "\tmovl %%eax,%%ss" : : : "eax");
336
337      /* Jump to the 16-bit code that we copied earlier.  It disables paging
338         and the cache, switches to real mode, and jumps to the BIOS reset
339         entry point. */
340
341      __asm__ __volatile__ ("ljmp $0x0008,%0"
342                    :
343                    : "i" ((void *) (0x1000 - sizeof (real_mode_switch) - 100)));
344  }
```

整个过渡的过程都不容许中断，所以一开始先关闭中断。然后向 CMOS 存储器中地址为 0x8f 处写入 0。注释中说这是 BIOS 所要求的，并且同时也起了关闭“不可屏蔽中断 ”NMI 的作用，我们就不加考证了。

==================== include/asm-i386/mc146818rtc.h 22 25 ====================

```
22  #define CMOS_WRITE(val, addr) ({ \
23  outb_p((addr),RTC_PORT(0)); \
24  outb_p((val),RTC_PORT(1)); \
25  })
```

以前，在向页式映射的过渡中，有一个时期需要在页面目录中的低区，即从虚地址 0 开始的区间也保持部分空间的映射，使得 CPU 以虚拟地址和物理地址访问内存时可以被映射到相同的物理存储单元。现在要向相反方向过渡也得要有这么一个时期，所以 283 行从页面映射目录 swapper_pg_dir 中把对应于系统空间的 256 个目录项复制到低区，放在从目录起点开始的地方。283～284 行中引用的常数分别定义于 include/asm-i386/pgtable-2level.h 和 include/asm-i386/pgtable.h 中：

==================== include/asm-i386/pgtable-2level.h 8 8 ====================

```
8   #define PGDIR_SHIFT    22
```

==================== include/asm-i386/pgtable.h 123 124 ====================

```
123  #define USER_PGD_PTRS (PAGE_OFFSET >> PGDIR_SHIFT)
124  #define KERNEL_PGD_PTRS (PTRS_PER_PGD-USER_PGD_PTRS)
```

这些目录项是从页面映射目录的高区原封不动复制下来的，可是系统空间的第一个页面、即 pg0[0]在正常运行中是无映射的，所以复制以后还要补上一个（见 289 行）。这里标志位_PAGE_RW 和_PAGE_PRESENT 的作用不言自明，而页面的基地址则为 0。实际上，这个页面正是现在要用的。

由于页面映射目录已经改变，294 行再装入一次控制寄存器%cr3。当然，因为是在内核中运行，原来%cr3 也是指向 swapper_pg_dir，装入前后这个地址并无改变，但是这条指令的执行使 CPU 重新装入页面映射目录。

如前所述，全局量 reboot_mode 的值是（当初）根据引导命令行中的选择项设置的。如果在命令行

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

中选择了“热引导”，那么这个变量的值就是 0x1234（见前面的 162 行），否则就是 0。BIOS 在执行中要测试内存单元 0x472 的内容，如果是 0 就要执行对内存等的自检，称为“热引导”；而若为 0 则跳过自检，称为“冷引导”。所以，302 行把 reboot_mode 的值写入这个内存单元，为 BIOS 的运行作好准备。

接下来（310～312 行），还要把两段小程序复制到物理内存第一个页面的顶部。这两个小程序就起着“台阶”的作用，一个是 real_mode_switch，另一个是作为参数传下来的 jump_to_bios。等一下读者就会看到这两个台阶是如何起作用的。

准备好台阶，就可以下了。首先（316 行）是把中断描述表换成 real_mode_idt，实际上是把所有的中断门和陷阱门都清除掉：

==================== arch/i386/kernel/process.c 205 205 ====================

```
real_mode_idt = { 0x3ff, 0 };
```

全局段描述表的改变就是关键性的了。322 行将新的全局段描述表 real_mode_gdt 装入 GDTR，其内容定义于 arch/i386/kernel/process.c：

==================== arch/i386/kernel/process.c 184 204 ====================

```
184  /* The following code and data reboots the machine by switching to real
185     mode and jumping to the BIOS reset entry point, as if the CPU has
186     really been reset.  The previous version asked the keyboard
187     controller to pulse the CPU reset line, which is more thorough, but
188     doesn't work with at least one type of 486 motherboard.  It is easy
189     to stop this code working; hence the copious comments. */
190
191  static unsigned long long
192  real_mode_gdt_entries [3] =
193  {
194      0x0000000000000000ULL, /* Null descriptor */
195      0x00009a000000ffffULL, /* 16-bit real-mode 64k code at 0x00000000 */
196      0x000092000100ffffULL  /* 16-bit real-mode 64k data at 0x00000100 */
197  };
198
199  static struct
200  {
201      unsigned short      size __attribute__ ((packed));
202      unsigned long long * base __attribute__ ((packed));
203  }
204  real_mode_gdt = { sizeof (real_mode_gdt_entries) - 1, real_mode_gdt_entries },
```

新的段描述项在数组 real_mode_gdt_entries[]中。对照第 2 章中对段描述项格式的定义，就可以看出，下标（即段选择号）为 1 的段描述项对应于代码段，其基地址为 0；而下标为 2 的段描述项则对应于数据段，其基地址为 100；两个段的大小都是 64KB。接着（330 行）把除 CS 以外所有的段寄存器都设置成 0x0010，即都是数据段。最后（341～343 行）是一条长程转移指令 ljmp（前缀“l”表示长距离）。跳转的目标怎样确定呢？段选择项是 0x0008，对应于 real_mode_gdt_entries[]中的第二项（195 行），即代码段，所以段的基地址为 0，而位移量则是（0x1000 - sizeof(real_mode_switch) - 100）。显然，

PDF-XCHANGE Click to buy NOW! www.docu-track.com
PDF-XCHANGE Click to buy NOW! www.docu-track.com

这就是前面复制好的台阶 real_mode_switch 的入口。这段程序以及 jump_to_bios 的机器代码以数据的形式定义于 arch/i386/kernel/process.c 中：

==================== arch/i386/kernel/process.c 226 243 ====================

```
226 static unsigned char real_mode_switch [] =
227 {
228     0x66, 0x0f, 0x20, 0xc0,           /*    movl  %cr0,%eax       */
229     0x66, 0x83, 0xe0, 0x11,           /*    andl  $0x00000011,%eax */
230     0x66, 0x0d, 0x00, 0x00, 0x00, 0x60,   /*    orl   $0x60000000,%eax */
231     0x66, 0x0f, 0x22, 0xc0,           /*    movl  %eax,%cr0       */
232     0x66, 0x0f, 0x22, 0xd8,           /*    movl  %eax,%cr3       */
233     0x66, 0x0f, 0x20, 0xc3,           /*    movl  %cr0,%ebx       */
234     0x66, 0x81, 0xe3, 0x00, 0x00, 0x00, 0x60, /*    andl  $0x60000000,%ebx */
235     0x74, 0x02,                   /*    jz    f             */
236     0x0f, 0x08,                   /*    invd                */
237     0x24, 0x10,                   /* f: andb  $0x10,al        */
238     0x66, 0x0f, 0x22, 0xc0            /*    movl  %eax,%cr0       */
239 };
240 static unsigned char jump_to_bios [] =
241 {
242     0xea, 0x00, 0x00, 0xff, 0xff          /*    ljmp  $0xffff,$0x0000  */
243 };
```

进入 228 行以后，由于 jmp 指令将与物理地址等同的线性地址装入了 IP，从此开始取指令的地址就都在低区了，这就是为什么需要在页面映射目录中先作好准备的原因。接着，通过改变控制寄存器%cr0 和%cr3 的内容将页面映射关闭。如果 CPU 的高速缓存还开着，则还要执行一条 invd 指令将高速缓存中的内容作废。然后，又通过改变%cr0 的内容回到实地址模式。执行完 238 行的 mov 指令，CPU 已经运行于实地址模式了。程序 jump_to_bios 的机器代码就紧接在 real_mode_switch[]的上方，所以执行完 238 行的 mov 指令以后紧接着就是 242 行的（长程）jmp 指令，目标是段地址为 0xffff，位移为 0 的地方，即 0xffff0，这就是 BIOS 的入口。